AFRL-IF-RS-TR-2000-48
Final Technical Report
April 2000

PARALLEL AND DISTRIBUTED ALGORITHMS FOR HIGH-SPEED IMAGE PROCESSING

University of Notre Dame

Robert L. Stevenson, Andrew Lumsdaine, Jeffery M. Squires,
and Michael P. McNally

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-2000-48 has been reviewed and is approved for publication.

APPROVED: [illegible]

TODD B. HOWLETT
Project Manager

FOR THE DIRECTOR: [illegible]

JOHN V. MCNAMARA, Technical Advisor
Information and Intelligence Exploitation Division
Information Directorate

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE APRIL 2000	3. REPORT TYPE AND DATES COVERED Final Jul 96 - Jan 98	
4. TITLE AND SUBTITLE PARALLEL AND DISTRIBUTED ALGORITHMS FOR HIGH-SPEED IMAGE PROCESSING			5. FUNDING NUMBERS C - F30602-96-C-0235 PE - 627022F PR - 4594 TA - 18 WU - PB
6. AUTHOR(S) Robert L. Stevenson, Andrew Lumsdaine, Jeffery M. Squires, and Michael P. McNally			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Notre Dame Department of Electrical Engineering Notre Dame IN 46556			8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFEC 32 Brooks Road Rome NY 13441-4114			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2000-48
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: Todd B. Howlett/IFEC/(315) 330-4592			
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Typical desktop workstations can be a severe bottleneck in the viewing and enhancement of imagery data. Due to the nature of many image processing algorithms, an effective method for alleviating this problem is through parallelism. Parallel hardware can come in many forms, from small clusters of workstations and workstations with many processors to dedicated hardware containing 10's, 100's and 1000's of processing nodes. One of the challenges is developing a portable parallel image processing library in such a potentially diverse environment. These issues led to the development of a Parallel Image Processing Toolkit (PIPT). The toolkit hides the detail of parallelization from the users of the PIPT and provides a uniform programming interface. In developing the toolkit the issues of advanced data handling, load, balancing and parallel visualization were addressed. In addition a specific computationally expensive High Resolution Video Stills algorithm was implemented within the PIPT.

The important contributions that resulted from this work transcend the individual tasks that were undertaken. First, the general approach that was taken to implement the PIPT illustrates several important principles for the design and implementation of general purpose parallel libraries. In this regard, the PIPT design can serve as a "design pattern" for an extensible parallel library. Second, the design pattern of the current implementation of the PIPT contains some notable attempts at programming image processing tasks in a generic fashion.

14. SUBJECT TERMS Image Processing, Distributed Computing, Parallel Processing, Message Passing Interface, MPI			15. NUMBER OF PAGES 168
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Air Force Research Laboratory Information Directorate AFRL/IF has for decades dealt with the needs of the intelligence community to acquire, analyze, and disseminate intelligence information. The Image Exploitation Processing Facility (IE-2000) is a research and development testbed for digital image processing. The IE-2000 facility develops and evaluates software and unique hardware to support imagery exploitation. To support its efforts, the IE-2000 undertook the development of an Image Processing Toolkit (IPT) in the summer of 1993. This toolkit provides a extensive library of basic image processing routines which can be used in a wide variety of image processing tasks. The toolkit was designed to be portable across many platforms. It was also designed so that additional functionally can be easily incorporated into the library at any time.

It has been recognized that the typical workstation does not have sufficient computational power to perform all the desired tasks needed in image exploitation. The computational problem is due to the very high resolution of the imagery data (typical image sizes can be as high as 10,000 by 10,000 pixels). Because of this, the processing power of the typical desktop workstation can become a severe bottleneck in the viewing and enhancement of the imagery data. Due to the nature of many image processing algorithms, an effective method for alievating this problem is through parallelism. Many image processing routines can achieve near linear speed-up with the addition of processing nodes. Parallel hardware can come in many forms, from small clusters of workstations and workstations with multiple processors (e.g., 4 processor UltraSPARC) to dedicated hardware containing 10's, 100's, or even 1000's of processing nodes. One of the challenges in developing a portable parallel image processing library in such a potentially diverse environment is the planning for not only the current platforms, but also for unknown future platforms.

These issues led to the development of the Parallel Image Processing Toolkit (PIPT) v1.0.3, sponsored by AFRL. To hide the details of parallelization from users of the PIPT, a registration/call-back mechanism was used to present a uniform programming interface. To achieve maximum performance and portability, the PIPT uses the Message Passing Interface (MPI) standard to effect parallelism. The leading implementations of MPI are in the public domain so that the PIPT can make use of this important standard while still remaining freely available itself.

1.1 Motivation

Although the PIPT v1.03 met the goals set out for it, important emerging technologies, such as symmetric multiprocessing and hierarchical memory systems, merited further investigation so that the Parallel Image Processing Toolkit can obtain the highest performance in these environments. Extending the PIPT to include the capabilities to exploit these technologies led to the development of the PIPT v2.1.

The work performed for this project (and reported in this document) concentrated on extending the performance and capabilities of the PIPT in the following areas:

- Advanced data handling
- Load balancing
- Parallel high resolution video stills
- Parallel visualization toolkit
- Adobe Photoshop interface

1.2 Work Performed

Advanced Data Handling Given the hierarchical structure of modern microprocessor memory systems, to gain maximum efficiency, algorithms must be cognizant of the costs associated with various types of memory accesses and be designed accordingly. As part of this effort, the core computational kernels of the PIPT were analyzed with respect to their use of hierarchical memory and were re-structured to make better use of it. Issues related to instruction pipelining were also studied. Finally, a thread interface for the PIPT was designed and implemented, providing multiple levels of available parallelism in the PIPT.

Load Balancing Heterogeneous clusters of workstations are one important target execution environment for the PIPT. In such an environment, where machines may have widely varying computational power as well as widely varying run-time loading, it is important for a parallel program to properly balance the jobs that are executed on the various nodes in the cluster. As part of this effort, a number of load balancing algorithms were designed, analyzed, and tested. The first-finish, first-served and the redundand first-finish, first-served both demonstrated effective load balancing in active heterogeneous clusters.

Parallel High Resolution Video Stills As an example of a high-end, computationally expensive task, the High Resolution Video Stills algorithm of Schultz and Stevenson [17] was parallelized using the PIPT.

Parallel Visualization One remaining serial bottleneck in a typical image exploitation task is in the final visualization stage. In a multiprocessor workstation with a shared memory architecture, it is possible to parallelize this final step as well. To this end, a parallel visualization library was developed and implemented.

Adobe Photoshop Interface To provide an effective, stable and familiar visualization environment, an interface between the Parallel Image Processing Toolkit and Adobe Photoshop was designed and implemented. Adobe Photoshop provides a public interface through which external programs can be incorporated (so-called plug-ins). An interface to the PIPT was incorporated as a filter plug-in.

1.3 Document Organization

This report is organized as follows. A review of parallel image processing and the parallel image processing toolkit is given in Chapter 2. Chapters 3, 4, 5, 6, and 7 provide detailed descriptions of the data handling, load balancing, parallel HRVS, parallel visualization, and Photoshop plug-in aspects of this effort, respectively. Chapter 8 summarizes the important lessons learned in this effort and makes some suggestions for future work.

The results from a comprehensive set of tests of the PIPT are given in Appendix A.

Chapter 2

Review of Parallel Image Processing and PIPT 1.0.3

2.1 Parallel Image Processing

The structure of the computational tasks in many low-level and mid-level image processing routines readily suggests a natural parallel programming approach. On either a fine- or coarse-grain architecture the most natural approach is for each computational node to be responsible for computing the output image at a spatially compact set of image locations. This generally will minimize the amount of data which needs to be distributed to the individual nodes and therefore minimize the overall communication cost. Thus, this approach will generally maximize the speedup of the parallel system. This section discusses the approach for data distribution utilized in the toolkit and the message passing system used to implement it.

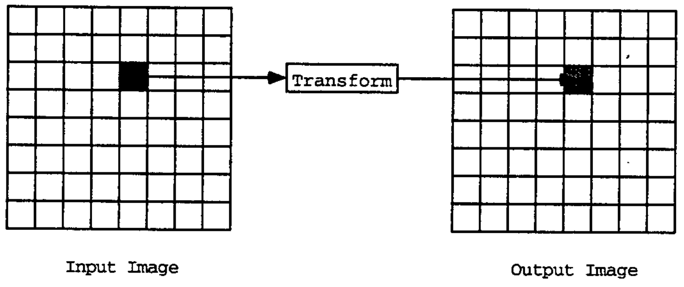


Figure 2.1: Data dependency for an image processing algorithm using a point operator.

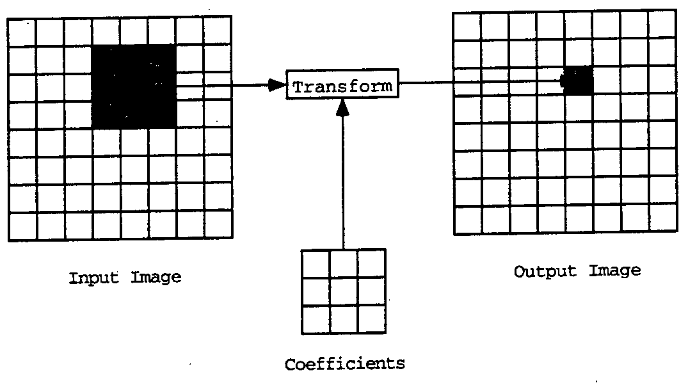


Figure 2.2: Data dependency for an image processing algorithm using a window operator.

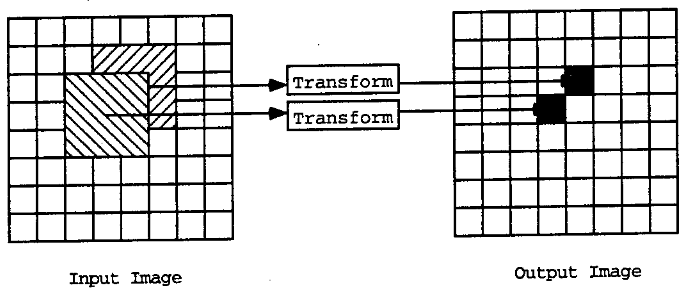


Figure 2.3: Fine-grained parallel decomposition for an image processing algorithm using a window operator. The parallel computation of two diagonally adjacent pixels is shown.

2.1.1 Data Distribution

Many image processing algorithms exhibit natural parallelism in the following sense: the input image data required to compute a given portion of the output is spatially localized. In the simplest case, an output image is computed simply by independently processing single pixels of the input image, as shown in Figure 2.1. More generally, a neighborhood (or window) of pixels from the input image is used to compute an output pixel, as shown in Figure 2.2. Clearly, the values of the output pixels do not depend on each other. Hence, the output pixels can be computed independently and in parallel. This high degree of natural parallelism exhibited by many image processing algorithms can be easily exploited by using parallel computing and parallel algorithms. In fact, many image processing routines can achieve near linear speedup with the addition of processing nodes (see Appendix A.4).

A fine-grained parallel decomposition of a window operator based image processing algorithm would assign an output pixel per processor and assign the necessary windowed data required for each output pixel to the corresponding processors. Each processor would perform the necessary computations for their output pixels. An example fine-grained decomposition of an input image is shown in Figure 2.3. A coarse-grained decomposition (suitable for MIMD or SPMD parallel environments) would assign

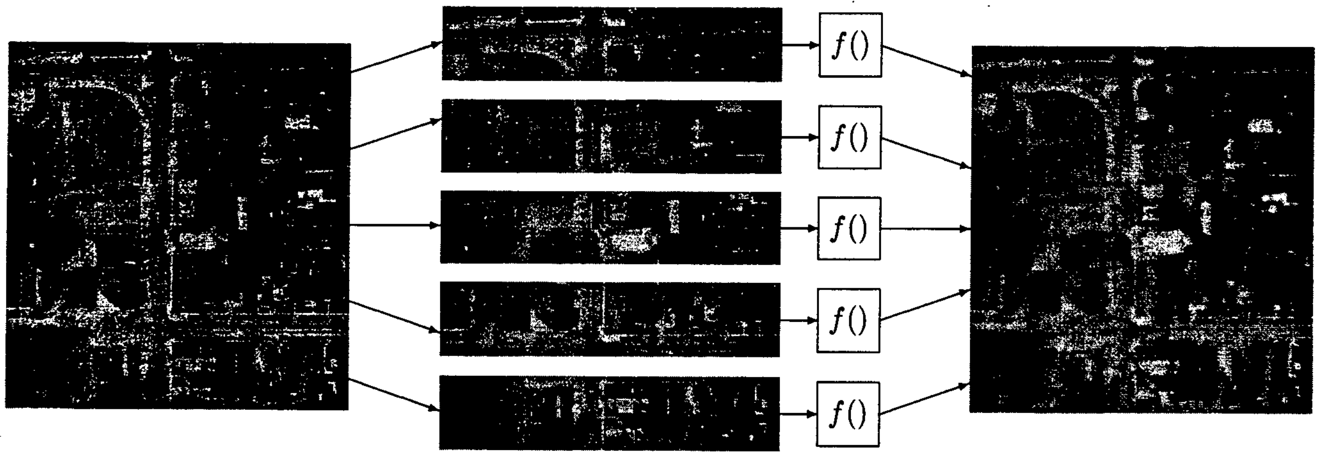


Figure 2.4: Coarse-grained parallel decomposition for an image processing algorithm using a window operator.

large contiguous regions of the output image to each of a small number of processors. Each processor would perform the appropriate window based operations to its own region of the image. Appropriate overlapping regions of the image would be assigned to properly accommodate the window operators at the image boundaries. An example coarse-grained decomposition of an input image is shown in Figure 2.4.

2.1.2 Message Passing

One of the challenges in developing a parallel image processing library is making it portable to the various (and diverse) types of parallel hardware that are available (both now and in the future). To make parallel code portable, it is important to incorporate a model of parallelism that is used by a large number of potential target architectures. The most widely used and well understood paradigm for the implementation of parallel programs on distributed memory architectures is that of *message passing*. Several message passing libraries are available in the public domain, including p4 [5], Parallel Virtual Machine (PVM) [3], PICL [10], and Zipcode [18]. Recently, a core of library routines (influenced strongly by existing libraries) has been standardized in the Message Passing Interface (MPI) [8, 11, 9]. Public domain implementations of MPI are widely available. More importantly, all vendors of parallel machines and high-end workstations provide native versions of MPI optimized for their hardware.

2.1.3 System Model

The PIPT, like the IPT before it, is constructed to allow the layered development of image processing applications. Figure 2.5 shows the system model for the original image processing toolkit. Applications use the IPT by calling image processing routines which are in turn built up from abstract computational kernel functions. New image processing routines are also built up from these kernel functions and then added to the IPT.

The PIPT uses a manager/worker scheme in which a manager program reads an image file from disk, partitions it into equally sized pieces, and sends the pieces to worker programs running on machines in the cluster. The worker programs invoke a specified image processing routine to process their sub-images and then send the processed sub-images back to the manager. The manager re-assembles the processed sub-images to create a final processed output image. Since the PIP Toolkit's internal image format stores rows of pixels contiguously in memory, sub-images are likewise composed of rows of pixels from the original image.

Figure 2.6 shows the system model for the PIPT. Applications use the PIPT by calling image processing routines which are in turn built up from abstract computation kernel functions. The kernel functions interface to an abstract transport layer which transparently effects parallel execution. The transport mechanism makes calls to a Message Passing Interface (MPI) library for its parallel communication operations. Although the PIPT provides for parallel execution of image processing routines, parallelism is encapsulated at a low level of the system so that users of the toolkit do not need to be concerned with parallel programming.

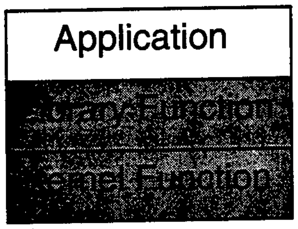


Figure 2.5: System model for applications using original image processing toolkit. The functionality provided by the original image processing toolkit is shown in grey.

A detailed diagram of the interaction between the various components of the PIPT is shown in Figure 2.7. The user application must provide a raw image as well as function parameters to the PIPT library function. The library function passes the raw image, the parameters, and a local processing function to a PIPT computational kernel. It is this local processing function which will be applied by the computational kernel on the worker nodes to actually process the image. In the C programming language, a function can be specified in this way by using a pointer to it. Unfortunately, in a distributed memory computing environment, a function pointer is not a meaningful way of specifying functions on remote compute nodes. Thus, each compute node constructs a translation table and stores the local memory addresses of necessary functions and data. Global (across all compute nodes) indices are then established for each function and for each variable. Thus, functions and variables can be specified remotely merely by sending the appropriate index as a message.

The transport mechanism uses MPI to pass slices of the image, as well as indices for looking up state

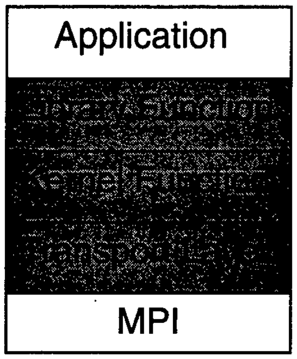


Figure 2.6: System model for applications using parallel image processing toolkit. The functionality provided by the parallel image processing toolkit is shown in grey.

information, to the worker nodes. The worker nodes use the indices to find and set values of local variables and then they process the image slice, passing back the image slice to the manager. The processed image is assembled by the transport mechanism and passed back to the computational kernel. The kernel passes the processed image back to the PIPT library function which in turn passes the processed image back to the user application.

2.2 PIPT 1.0.3

To support its efforts, the IE-2000 undertook the development of the Image Processing Toolkit (IPT) in the summer of 1993. The IPT provides an extensive library of basic image processing routines that can be used in a wide variety of image processing tasks. The toolkit was designed to be portable across many platforms. It was also designed so that additional functionality can be easily incorporated into the library at any time.

While the original IPT provides the necessary functionally and flexibility required for the intended tasks, it became clear that for use in an interactive workstation setting, many important image processing tasks are too slow. This is to be expected, given the large image sizes (commonly up to $10{,}000 \times 10{,}000$) and the complex operations which are required in image exploitation. Consequently, Parallel Image Processing Toolkit (PIPT) was developed, based on the original IPT, to allow for parallel execution of image processing tasks.

Like its predecessor (the IPT), the PIPT is easily extensible and provides a programming interface that largely hides parallelism from the user. Inside the Toolkit, a message-passing model of parallelism is designed around the Message Passing Interface (MPI) standard. In a typical workstation cluster, and in a dedicated parallel environment, the PIPT was able to obtain nearly linear speedup with respect to the number of processors on typical image processing tasks.

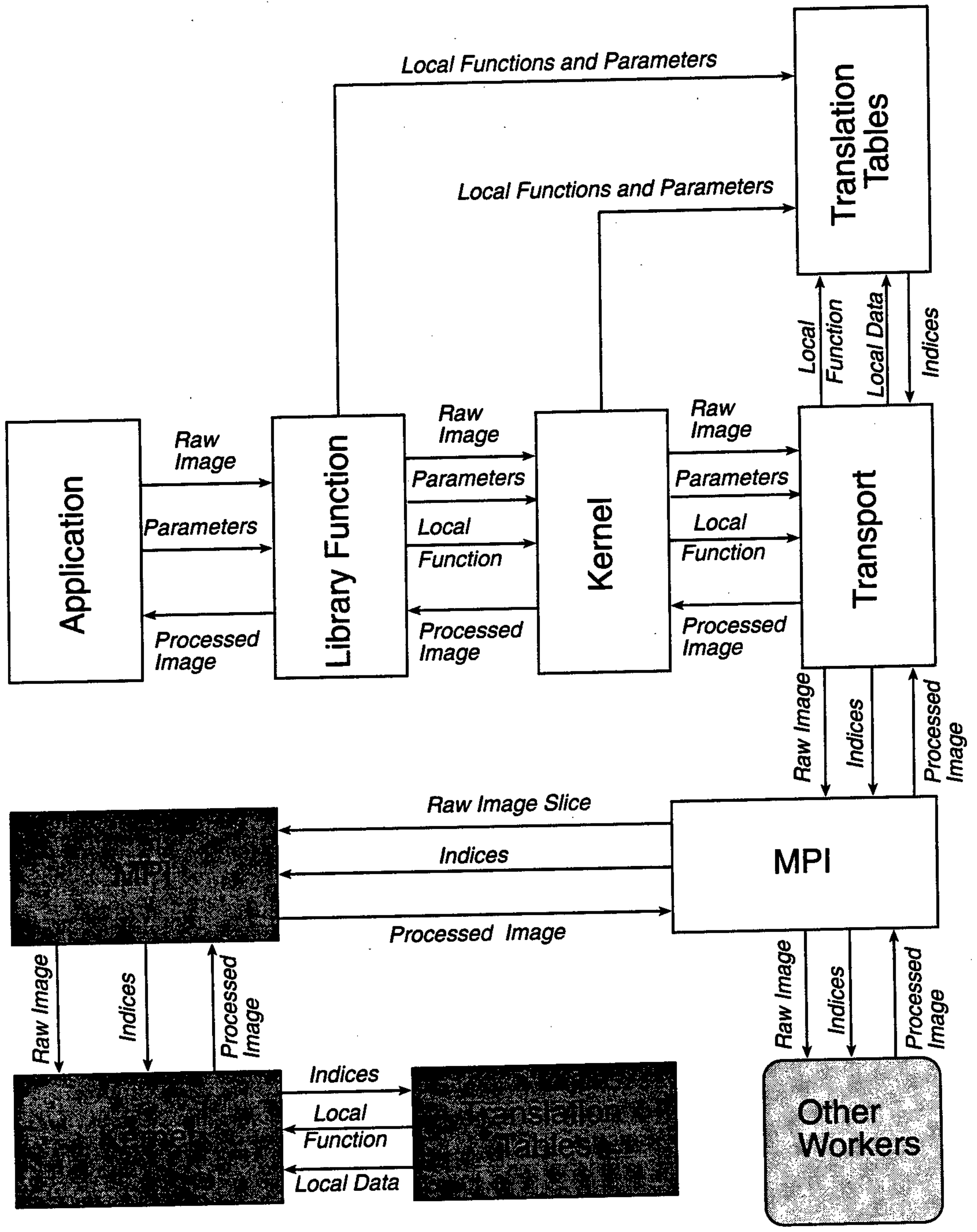


Figure 2.7: Detailed diagram of the interaction between the various components of the PIPT. Components located on worker processors are shown in grey.

Although the PIPT v1.0.3 achieved its original project goals, there were still some important open issues regarding parallel image processing.

1. Extending the PIPT to obtain high performance on symmetric multiprocessors,
2. Achieving high performance in the presence of non-uniform memory access environments,
3. Load balancing for maximum parallel efficiency, and
4. Parallel visualization.

We discuss each of these issues below. More details will be given in Chapters 3, 4, 6 regarding their actual implementation in the PIPT v2.0.

Shared Memory

The PIPT was designed in part to be able to exploit cluster-based parallelism, since this is a ubiquitous and cost-effective parallel computing environment. More recently, symmetric multiprocessors (SMPs) have emerged as an alternate, but perhaps complementary, parallel computing resource.

Since the PIPT was developed using MPI, it is portable to most parallel computing environments, including SMPs. In fact, the PIPT v1.0.3 has already been ported to at least two high-end SMP architectures, the SGI Power Challenge and the Cray J90. There is, however, an inherent inefficiency in v1.0.3's approach when the parallel processing is performed on an SMP architecture. The PIPT v1.0.3 uses a master processor which maintains the entire image data set and passes out segments of the data to individual CPUs for processing, as shown in Figure 2.8. This can be easily mapped onto an SMP architecture by segmenting the single shared memory resource between CPUs. The master CPU still maintains the only complete copy of the image data and still passes out segments of data to individual CPUs, as shown in Figure 2.9. However, there is no need for this extra data copy on an SMP architecture since all of the CPUs can access the main image data set. Thus, we modified the PIPT to take advantage of shared memory parallelism through the use of multi-threading while still using message passing for distributed memory environments. Thus, on a single SMP workstation a master CPU will still allocate work to individual processors, but the message that is passed between processors is a small control message and not a segment of the image data, as shown in Figure 2.10. However, due to the lack of thread-safe MPI imeplementations, while this scheme has been implemented on the workers, it is not possible to implement it on the manager; the exact problem is discussed in Section 3.7.3.

The PIPT will thus incorporate a hierarchical model of parallelism that will allow it to function on a single workstation, a single SMP, or a cluster of workstations and/or SMPs, see Figures 2.11 and 2.12. Although the PIPT contains a large number of routines, these routines rely on a relatively small number of computational kernels. By concentrating our efforts on multi-threading these kernels, we will be able to provide hierarchical parallelism to the entire toolkit with only modest effort (a similar approach

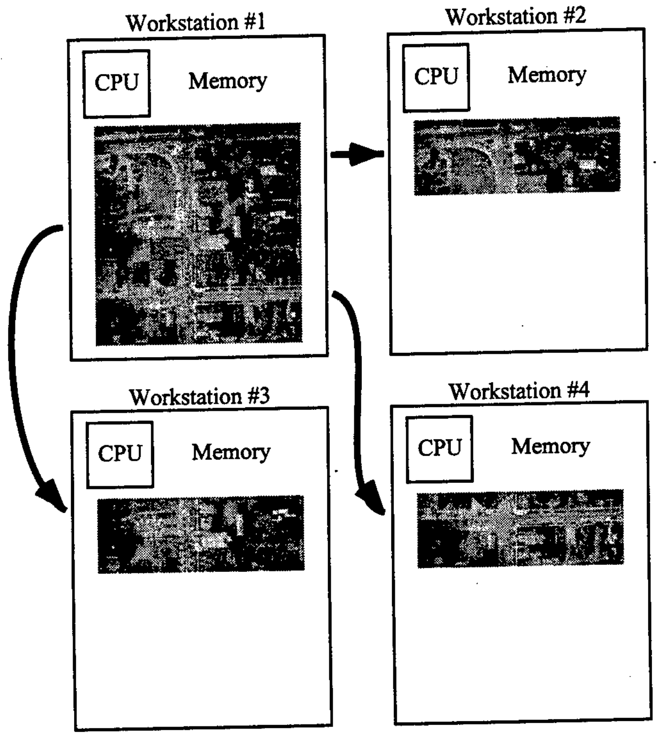


Figure 2.8: Data flow in PIPT 1.0.3.

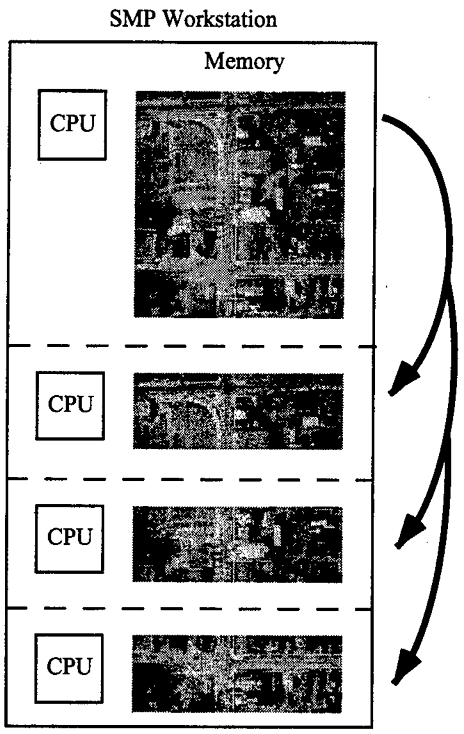


Figure 2.9: Data flow in PIPT 1.0.3 on an SMP architecture

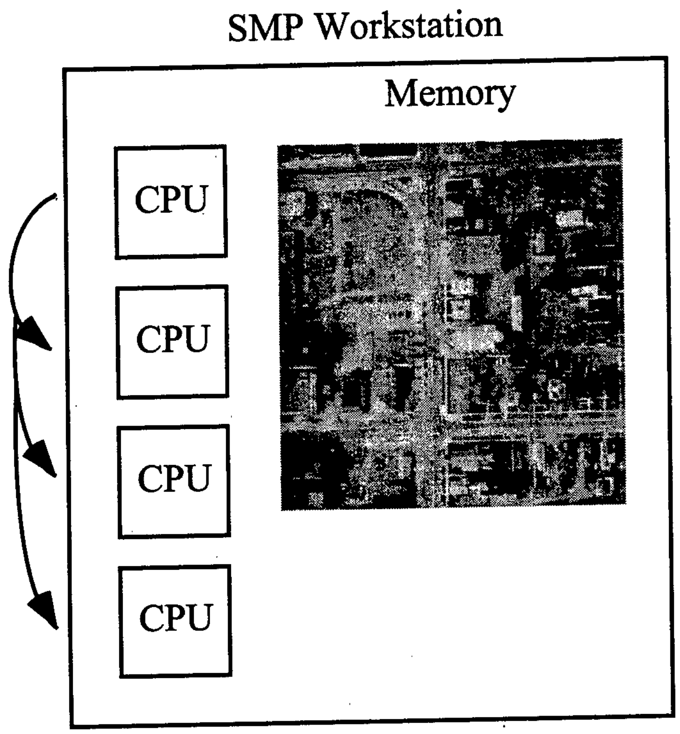


Figure 2.10: Proposed data flow in PIPT 2.1 on a SMP workstation.

was used in parallelizing the original IPT to produce the PIPT, see [15, 20]) and maintain a consistent programming interface.

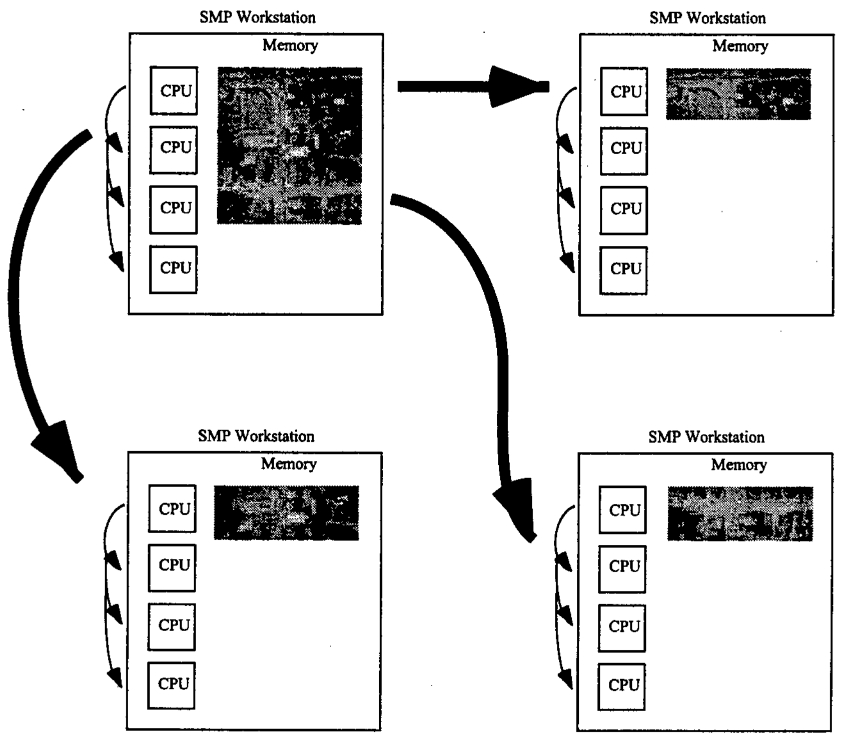


Figure 2.11: Data flow in PIPT 2.1 on a SMP workstation cluster.

Non-Uniform Memory Access

Image processing requires not only a large number of computations but a large amount of data movement as well. If a program is not careful, the costs of data movement can easily overwhelm the costs of computation.

One important design feature of modern RISC microprocessor-based workstations is a hierarchical memory system. That is, the microprocessor can operate at maximum efficiency with data in registers and in cache, but at a much lower level of performance for data in main memory, or even worse, swapped out to disk. This is due to the fact that modern RISC microprocessors only require a single clock cycle to execute one or more instructions with en-registered data. Moving a word of data from cache to a register normally takes a single clock cycle, but moving data from main memory to cache takes several clock cycles. Thus, if data is not en-registered or not in cache, the microprocessor must simply wait until the data can be moved into cache. Figure 2.13 illustrates a typical hierarchical memory system.

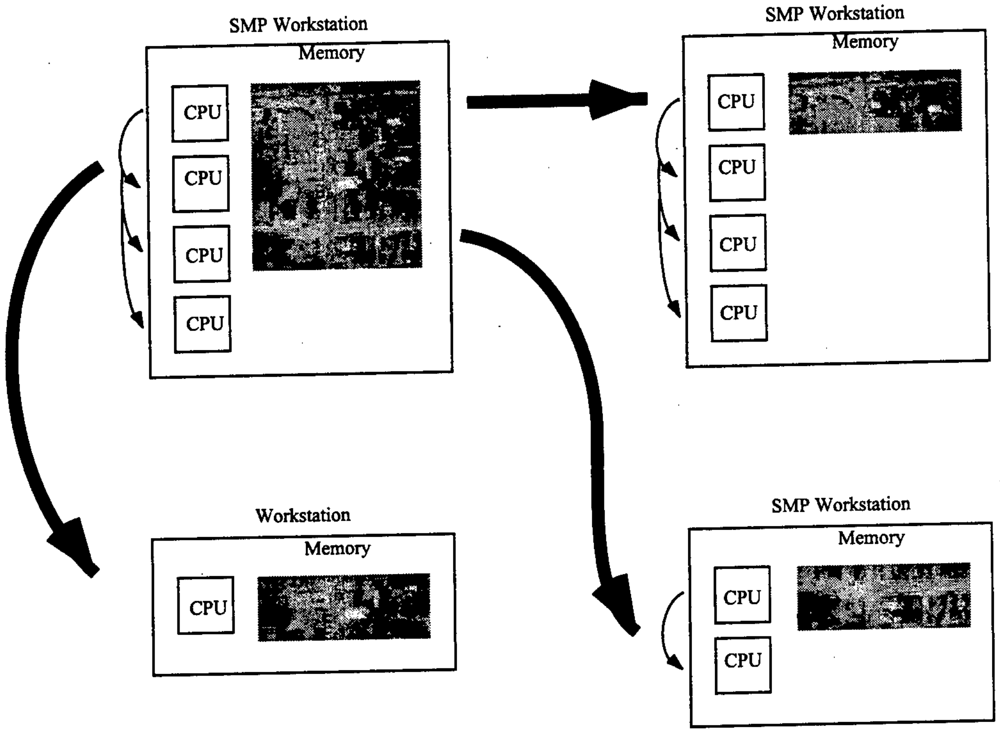


Figure 2.12: Data flow in PIPT 2.1 on a non-heterogeneous workstation cluster.

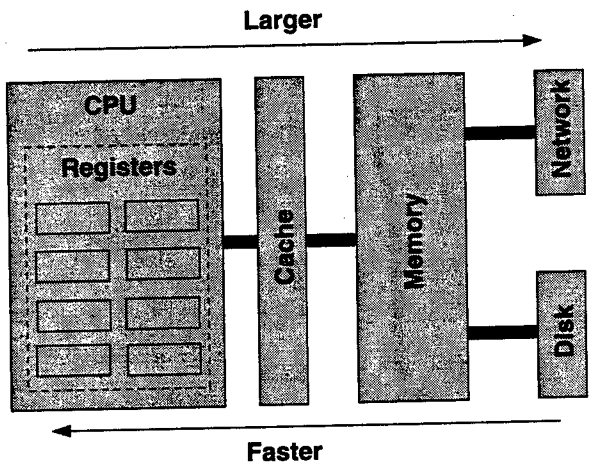


Figure 2.13: Hierarchical memory system.

To obtain highest computational performance on a modern microprocessor, it is crucial for computationally intensive problems to take the hierarchical nature of the memory system into account. The fundamental task is to minimize data movement to and from main memory — once moved to cache, data should be kept there and used for as long as possible. Typical strategies for increasing memory system performance focus on unrolling and blocking loops, restructuring algorithmic memory access patterns, removal of unnecessary dependencies in code blocks, and so forth [4].

Optimizing an algorithm for maximum memory efficiency can have a dramatic effect on performance. Table 2.1 shows the MFLOP rates obtained while performing matrix-matrix multiplies on an RS/6000 590, using several different blocking strategies in the code. The peak MFLOP rate for this machine is 266.67 MFLOP/sec, which requires that four floating point operations be performed on each clock cycle. The best blocking scheme achieves 96.6% of the peak. Note that a poor choice of blocking scheme gives very sub-optimal code, achieving only 38.7% of peak — a very poor usage of CPU.

MFLOP rate	Matrix size	M_0	K_0	N_0	% peak
103.3	100	1	1	2	38.7
168.6	100	1	1	4	63.0
257.5	100	1	1	10	96.5
102.3	100	1	2	2	38.4
170.6	100	1	2	4	64.0
180.3	100	1	2	10	71.4
157.0	100	2	2	2	58.9
228.4	100	2	2	4	85.6
133.6	100	2	2	10	50.1

Table 2.1: MFLOP rates obtained on RS/6000 Model 590 for matrix-matrix multiply using various blocking strategies (M_0, K_0, N_0 represent cache-register blocking parameters).

An important aspect of this work will be to study the issues of hierarchical memory and to develop general techniques for maximizing the performance of image and video processing algorithms in the presence of non-uniform memory access. The computational structure of matrix-matrix multiplication is very regular, as is that of many image and video processing tasks. Many of the techniques used to obtain near-peak performance for matrix-matrix multiplication should be applicable to image and video processing.

Load Balancing

In order to extract maximum performance and efficiency from a parallel computing environment, it is critical that the processing load be properly distributed among the processing nodes. Since the entire parallel computation cannot be completed until all nodes have completed their computations, the parallel computation will be limited by the slowest processor. A single processor taking longer to complete its task than the other processors will hold up the entire parallel computation. In an ideally distributed parallel computing task, all the processing nodes will complete their tasks at precisely the same time. The goal of load balancing is to partition the parallel task in such a way that this will occur.

Load balancing becomes particularly difficult in an environment such as a heterogeneous workstation cluster because:

- In a heterogeneous cluster, different processing nodes may have different raw computing power.
- In an active workstation cluster, different processing nodes may be loaded by running other tasks.

There are several approaches that can be taken to load-balancing.

1. Partition the parallel task into a large number of relatively small tasks such that there are many more tasks than processors. Allow the worker processors to request processing tasks on a first-come first-serve basis.
2. Actively monitor the available processing power of the processing nodes being used and adjust the partitioning of tasks to achieve optimal balance.

The first approach was incorporated into the PIPT 1.0.3 with encouraging results (see [15, 20]). This approach has the disadvantage that it increases the number of communication operations that must be performed, thus increasing the total communication cost. In some situations, this might be outweighed by the savings gained by good load-balancing, but in other situations it might not. Further refinements, such as adjusting the granularity of the task division, or combining this approach with the second, are discussed in Chapter 4.

Parallel Visualization

One vital and time-consuming portion of the image exploitation process is the final rendering of an image onto the computer monitor. This is due to relatively computationally complex operations of scaling, rotating, and color mapping that typically takes place. With a standard visualization package all of this processing is done with a single process on a single CPU, as shown in Figure 2.14. As part of this work, we explored various methodologies (through multithreading) for accelerating the visualization process with parallel processing on SMP workstations. The resulting visualization package efficiently utilizes the the available compute cycles on all of the available CPUs, as shown in Figure 2.15. Through the combination of the parallel image processing toolkit and the parallel visualization toolkit, image data is both processed and displayed efficiently. This results in the quickest turnaround from the request for an operation until the resultant image is viewed.

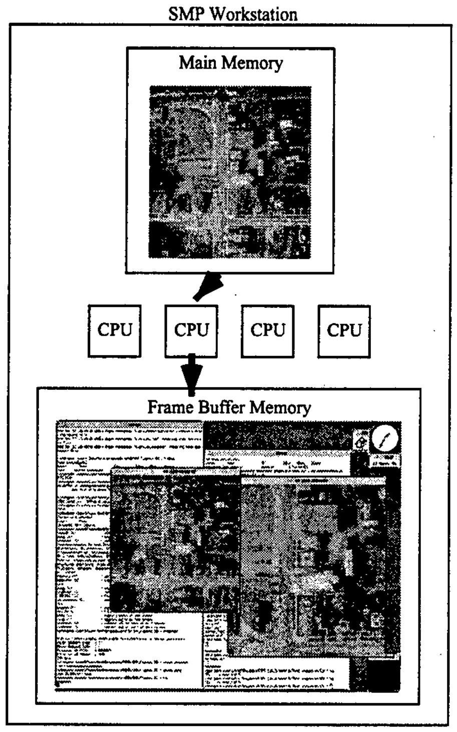


Figure 2.14: Data flow and CPU usage with a standard visualization package

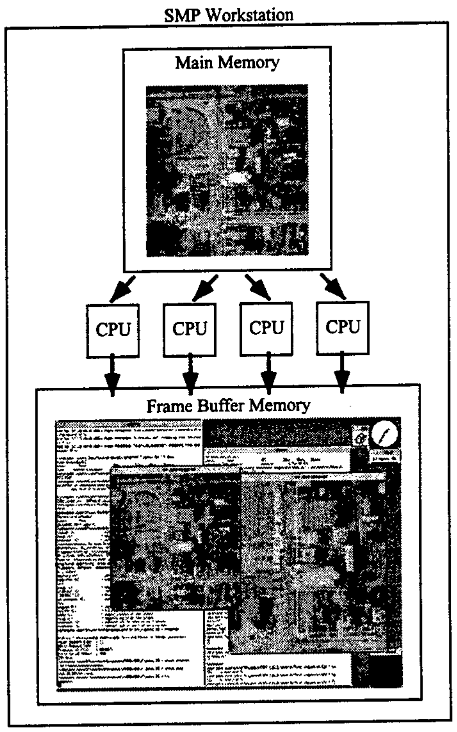


Figure 2.15: Data flow and CPU usage with a parallel visualization package

Chapter 3

Advanced Data Handling & Shared Memory

3.1 Overview

In the PIPT, there is not only a large amount of computation related to image processing, but also a large amount of data movement both between the CPU and main memory, and between main memory and the underlying communication network.

Architectural advances that have allowed microprocessors to attain their high level of performance include hierarchical memory and pipelining. Most modern microprocessor-based workstations use hierarchical memory systems to alleviate the high cost of main memory access (relative to computation) [12]. Because of this high cost, data movement must be handled efficiently or the cost of data movement will overwhelm the costs of computation.

A hierarchical memory system typically consists of five parts: registers, multiple levels of cache, memory, disk, and network (see Figure 3.1). A RISC-based microprocessor can only operate on data in registers but is able to load and store data from cache to register in a single clock cycle. If data is not in the cache, a load operation will require the CPU to wait until the data can be moved into the cache — a process that can take numerous clock cycles. The worst case in a virtual memory system is that the data is moved from disk to memory or from memory to disk (the time depends on the mechanical response time).

3.2 Cache

There are three types of cache misses:

Compulsory If the first access to a datum is not in cache, data is brought in to cache in line size pieces. There is nothing that the programmer can do about this type of cache miss, although the latency of the transfer can be alleviated though pre-fetching.

Capacity If the cache is too small to store the data used during execution, blocks will be discarded to make room for new blocks. A miss occurs when a block that was previously in the cache has been discarded and later retrieved.

Conflict If the block placement strategy is set associative or direct mapped, conflict misses will occur when two blocks are assigned to the same cache location. This may occur even if there is enough room in the cache to hold both blocks.

To increase performance, it is important that the PIPT takes advantage of the hierarchical memory system: data should be kept in cache as long as possible to reduce capacity misses.

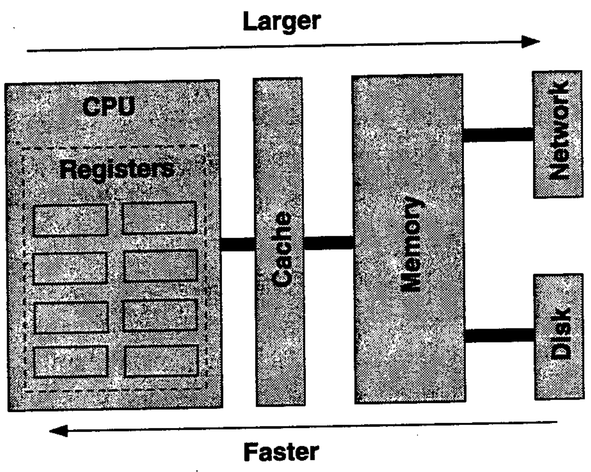


Figure 3.1: A typical hierarchical memory system.

3.3 Pipeline

RISC-based microprocessors typically have a multi-stage instruction pipeline. Pipelining is a technique whereby multiple instructions may be overlapped during execution so that one or more instructions are completed per clock cycle. A pipeline is like an assembly line; each stage of the pipeline contributes to the execution of an instruction.

The number of pipeline stages varies from processor to processor, however there are five main stages: instruction fetch (IF), instruction decode/register fetch (ID), execute/address calculation (EX), memory access (MEM), and write back (WB) [12].

Ideally, the speedup from pipelining equals the number of pipeline stages. Since the un-pipelined machine executes each instruction in a single clock cycle, its average time per instruction is simply the clock cycle time. With a pipelined machine, the clock cycle time is reduces by a factor roughly equal to the number of stages. Figure 3.2 demonstrates an ideal pipeline. Usually however, the stages will

Instruction	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Figure 3.2: Ideal pipeline

not be perfectly balanced, and the pipelining structure introduces some overhead. In addition, there are situations called *hazards* that prevent a stage of the pipeline from doing work. Specifically, there are three types of hazards:

Structural Hazards These hazards occur when the hardware cannot support the combination of instructions in simultaneous overlapped execution. For example, a structural hazard occurs when two stages of the pipeline require the same hardware resource.

Data Hazards Data hazards arise when one instruction depends on a result that has not yet been made available by a previous instruction.

Control Hazard Control hazards arise from the pipelining of branches and other instructions that change the program counter. Control hazards can sometimes be avoided by using a branch prediction algorithm (which must be implemented in the hardware itself).

When a hazard is encountered, a *stall* is introduced into the pipeline; for one or more clock cycles, no more instructions are introduced into the pipeline. When the hazard has been completed, new instructions are issued into the pipeline, and execution continues normally. The actual speedup from pipelining is the pipeline depth divided by one plus the number of stall cycles per instruction. Performance can be improved by reducing the number of data and control hazards. Section 3.5 describes techniques for doing this.

Figure 3.3 demonstrates a data hazard. The first instruction loads a value from memory into register one. The second instruction subtracts the contents of register five from register one, and puts the answer in register four. The MEM cycle of the load produces a value that is needed in the EX cycle of SUB, which occurs at the same time. The problem is solved by inserting a stall [12]. The following sections illustrate different techniques that can be used to reduce the number of cache misses or pipeline stalls. Each technique is presented with an example that compares the unoptimized method with the technique discussed in that section. The code fragments used in the following sections have been compiled with the KCC compiler from Kuck and Associates [14]. The codes are compiled with no optimizations to

Instruction	1	2	3	4	5	6	7	8	9
LW R1,0(R1)	IF	ID	EX	MEM	WB				
SUB R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR R8,R1,R9				stall	IF	ID	EX	MEM	WB

Figure 3.3: Pipeline with a data hazard

more clearly demonstrate the method being described. Next, performance results with full optimization (`-fast -O4 +K3`) are provided to demonstrate what the compiler can (and cannot) do to optimize the code. The result provided with each example is the best wall clock time of five trials on an unloaded machine.

3.4 Reducing Cache Misses

Reducing cache misses is extremely important to improving the performance of a given program. Loading data from memory is unavoidable, so the cost of the initial cache miss is unavoidable. But once the data has been loaded from memory, it should be operated on as long as possible while it is still in the cache. This will avoid repeatedly paying the high cost of memory access.

3.4.1 Loop Interchange

The order of nested loops can affect memory access patterns. For example, when accessing data from the two dimensional array shown in Figure 3.4, simply exchanging the nesting of the loops can make the code access the data sequentially rather than striding through memory (Figure 3.5). Table 3.1 shows how reordering the loops maximizes use of data in a cache before it is discarded.

The original code would skip through memory in strides of N words. This is the "wrong" loop order. Since all the data cannot be loaded into the cache at once for large problem sizes, cache misses occur frequently. For the optimized version (the "right" loop order), the array values are accessed sequentially through memory, accessing all the words in the cache block before going to the next outer loop iteration. This optimization improves cache performance by increasing the hit rate of the cache without affecting the number of instructions executed.

The results are as expected. Notice the discontinuity between when the loops are ordered the wrong way between N=256 and N=512.

In this example, the data associated with the two dimensional array x is contiguous in memory. In C and C++, the first index refers to the major dimension. For this example, $x[i][j]$ is adjacent to $x[i][j+1]$

in memory, and $x[i][j]$ is N locations away from $x[i+1][j]$ in memory.

```
double **x;
// Allocate x to be a contiguous C style 2D array

for (j = 0; j < N; ++j)
  for (i = 0; i < N; ++i)
    x[i][j] = 2 * x[i][j];
```

Figure 3.4: Loops in the "wrong" order

```
double **x;
// Allocate x to be a contiguous C style 2D array

for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    x[i][j] = 2 * x[i][j];
```

Figure 3.5: Loops in the "right" order

In the PIPT, all nested loops were examined to ensure that they were in the right order. This included a code review of all routines and kernels. Some of the original IPT routine loops were found to be incorrectly ordered, and were corrected.

3.4.2 Loop Fusion

Loop fusion is a method that can be used when a program uses two or more distinct loops over the same arrays with the same loop indices. By combining the loops together into a single loop, the data fetched into the cache can be used repeatedly before being swapped back out to memory. This increases the temporal locality of the data, which improves the cache behavior. In Figure 3.6, the original code generates cache misses accessing the z array in *both* loops; the cache miss penalty is paid twice. This happens because for large enough N, the entire z array cannot fit in the cache; capacity misses near the end of the first loop cause more capacity misses through the second loop. In the fused loop shown in Figure 3.7, the number of cache misses is essentially cut in half because both z accesses can utilize a single cache miss. Table 3.2 shows how loop fusion increases performance. While the differences in wall clock time shown in the table may seem small, taken over many iterations of repeated calculations, the aggregate time difference can be significant.

There were several smaller loops in various PIPT utility routines and within the opaque transport mechanism that benefited from being fused. It is difficult to quantify their exact speedup; since the changes

	No optimization		Full optimization	
Size (N)	Wrong order	Right order	Wrong order	Right order
4	0.000005	0.000257	0.000001	0.000002
8	0.000037	0.000907	0.000003	0.000003
16	0.000083	0.000280	0.000009	0.000005
32	0.000327	0.000269	0.000034	0.000018
64	0.001076	0.000972	0.000137	0.000060
128	0.004565	0.004020	0.001101	0.000237
256	0.017709	0.016046	0.004414	0.000839
512	0.133931	0.066672	0.103657	0.006921
1024	0.628488	0.267148	0.482482	0.028425

Table 3.1: Loop interchange performance results

```
double **x, **y, **z;
// Allocate x, y, and z
// to be contiguous C style 2D arrays

for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    x[i][j] += z[i][j];

for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    y[i][j] += z[i][j];
```

Figure 3.6: Without loop fusion

```
double **x, **y, **z;
// Allocate x, y, and z
// to be contiguous C style 2D arrays

for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j) {
    x[i][j] += z[i][j];
    y[i][j] += z[i][j];
  }
```

Figure 3.7: Optimized loop fusion example

	No optimization		Full optimization	
Size (N)	Simple	Fused	Simple	Fused
2	0.000001	0.000001	0.000001	0.000001
4	0.000002	0.000001	0.000001	0.000002
8	0.000004	0.000005	0.000004	0.000005
16	0.000014	0.000016	0.000013	0.000016
32	0.000082	0.000079	0.000096	0.000086
64	0.000330	0.000376	0.000339	0.000387
128	0.001299	0.001494	0.001312	0.001507
256	0.012890	0.011536	0.013024	0.011536
512	0.053770	0.046722	0.053608	0.046667
1024	0.211119	0.188688	0.209804	0.188558

Table 3.2: Loop fusion performance results

were in the PIPT engine itself, all routines were affected depending on how many parameters they registered, which hook routines were called, etc.

3.4.3 Pointer Dereferencing

Pointer dereferencing can be an expensive operation. Multi-dimensional array dereferencing is a common source of performance penalties, particularly when it is the central operation of nested loops. Figure 3.8 shows a nested loop that assigns a three dimensional array value from another three dimensional array value. This statement causes eight pointer dereferences. First, the `x` pointer must be resolved to find the array of planes. The `i`th element is then located and dereferenced to find the array of rows. The `j`th element is similarly found and dereferenced to find the array of columns. Finally, the `k`th element is located so that it can be assigned to. The dereferencing of `y` is analogous. The total number of dereferences is $8N^3$.

The optimized code in Figure 3.9 amortizes the cost of pointer dereferences by extracting pointer references up to the level of the loop where they are actually changed. For example, by moving the plane array dereference to the outer loop, it is dereferenced N times instead of N^3 times. Similarly, the row pointer deference is moved to the middle loop, leaving only the specific element dereference in the innermost loop. The total number of dereferences is $2(N + N^2 + N^3)$, which, for large N, is $\ll 8N^3$. Figure 3.3 shows the timing results between the simple dereference and the amortized dereference.

Every PIPT kernel has several nested loops similar to Figure 3.8 that loops over the planes of pixels in the input image/feature(s). Additionally, several routines loop over images or planes of pixels. All PIPT code that had nested loops over multi-dimensional arrays were restructured to amortize the cost of the pointer dereference. The `IPWindowMoments()` routine benefitted from amortized pointer dereferencing; a comparison of execution times from the PIPT 1.0.3 and 2.1 is shown in Table 3.4.

```
int ***x, ***y;
// Allocate x and y to be contiguous C style 3D arrays

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      x[i][j][k] = 2 * y[i][j][k];
```

Figure 3.8: Pointer dereferencing example

```
int ***x, **x_plane, *x_row;
int ***y, **y_plane, *y_row;
// Allocate x and y to be contiguous C style 3D arrays

for (i = 0; i < N; i++) {
  x_plane = x[i];
  y_plane = y[i];
  for (j = 0; j < N; j++) {
    x_row = x_plane[i];
    y_row = y_plane[i];
    for (k = 0; k < N; k++)
      x_row[k] = 2 * y_row[k];
  }
}
```

Figure 3.9: Optimized pointer referencing examples

	No optimization		Full optimization	
Size (N)	Simple	Amortized	Simple	Amortized
2	0.000766	0.000090	0.000764	0.000090
4	0.000158	0.000094	0.000157	0.000094
8	0.000207	0.000123	0.000202	0.000124
16	0.005097	0.000339	0.006023	0.000385
32	0.020339	0.002160	0.019513	0.002337
64	0.089070	0.015028	0.085139	0.015035
128	0.456674	0.112652	0.436601	0.110387
256	2.595699	0.869196	2.367717	0.811176

Table 3.3: Amortized pointer dereferencing performance results

Configuration	**Device**	**Basetime**	**Newtime**	**Speedup**
1 Sun UltraSPARC 140e	N/A	14641.126	10477.192	1.397
2 Sun UltraSPARC 140e	10bT	11076.523	5910.164	1.874
4 Sun UltraSPARC 140e	10bT	7538.262	3185.516	2.366
8 Sun UltraSPARC 140e	10bT	4654.274	1802.212	2.583
2 Sun UltraSPARC 140e	100bT	10385.549	5480.389	1.895
4 Sun UltraSPARC 140e	100bT	6387.522	2964.529	2.155
8 Sun UltraSPARC 140e	100bT	3546.472	1708.583	2.076

Table 3.4: Comparison of `IPWindowMoments()` in PIPT 1.0.3 and 2.1, with parameters window height = 9, window width = 9, and number of moments = 7. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

3.4.4 Array Reference Copying

Copying an entire array can be expensive, particularly for large arrays. In general, array copies can be avoided by passing a pointer to the beginning of the array; this passes the array by *reference* rather than by *value*. Even when passing multi-dimensional arrays, although multiple pointers must be copied to pass by reference, the cost is still less than passing by value.

Figure 3.10 shows a two dimensional array copy by value. Although the cost of the copy itself is large, if both arrays do not fit in the cache, the copied array will generate capacity cache misses when it is accessed later. Figure 3.11 shows a two dimensional array copy by reference. In this case, only the row pointers need be copied. Copying multi-dimensional arrays by reference only requires the lowest level pointers to be copied, or N^{d-1} pointers, where d is the number of dimensions in the matrix). Copying by value requires copying all the data, which is N^d copies. Table 3.5 shows the timing results for both cases.

```
int **x, **y;
// Allocate x and y to be contiguous C style 2D arrays

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    x[i][j] = y[i][j];
```

Figure 3.10: 2D array copy by value example

Several PIPT image processing routines require the input pixel window to be stored row-wise in contiguous memory (see Figure 2.2 in Section 2.1.1). This class of routines typically involves a sorting algorithm. As such, each window of pixels passed to the window operator function must be copied by value from the input image to a pre-allocated window. This copy, since it is performed for every pixel

```
int **x, **y;
// Allocate x to be a contiguous C style 2D array
// Allocate y to be a "pointer skeleton" array, but do not
// allocate any memory for the actual data

for (i = 0; i < N; i++)
  x[i] = y[i];
```

Figure 3.11: 2D array copy by reference example

	No optimization		Full optimization	
Size (N)	Value	Reference	Value	Reference
2	0.000822	0.000089	0.000818	0.000089
4	0.000145	0.000088	0.000146	0.000088
8	0.000142	0.000088	0.000144	0.000089
16	0.000160	0.000088	0.000163	0.000090
32	0.000213	0.000089	0.000213	0.000091
64	0.000415	0.000091	0.000466	0.000091
128	0.001149	0.000094	0.001078	0.000094
256	0.004066	0.000102	0.003779	0.000102
512	0.015756	0.000116	0.014320	0.000190
1024	0.061220	0.000142	0.055736	0.000136

Table 3.5: Copy by value / copy by reference examples

window passed to the window operator function, is quite expensive.

But many other routines do not require the pixels to be stored in contiguous memory. For this class of routines, the window copy can be performed by reference. This can result in a substantial speedup, particularly for routines that are not compute intensive. Table 3.6 shows a performance comparison of of the PIPT 1.0.3 (which uses array value copies) and PIPT 2.1 (which uses array reference copies).

Configuration	**Device**	**Basetime**	**Newtime**	**Speedup**
1 Sun UltraSPARC 140e	N/A	161.816	38.188	4.237
2 Sun UltraSPARC 140e	10bT	92.636	31.116	2.977
4 Sun UltraSPARC 140e	10bT	49.435	19.808	2.496
8 Sun UltraSPARC 140e	10bT	27.855	13.321	2.091
2 Sun UltraSPARC 140e	100bT	81.814	23.441	3.490
4 Sun UltraSPARC 140e	100bT	41.292	13.706	3.013
8 Sun UltraSPARC 140e	100bT	21.315	8.182	2.605

Table 3.6: Comparison of `IPAverage()` in PIPT 1.0.3 and 2.1, with parameters window height = 11 and window width = 11. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

3.5 Reducing Pipeline Stalls

Pipeline architectures rely on instruction-level parallelism to achieve high performance. The number of hazards is reduced when instructions are are not dependent on each other, which allows them to be executed simultaneously in the pipeline. A simple and common technique to increase the amount of instruction-level parallelism available in a program is to exploit parallelism within the iterations of a loop (loop level parallelism). The following techniques can be applied to reduce pipeline hazards.

3.5.1 Loop unrolling

The main idea behind loop unrolling is to do more work per loop iteration such that the cost of branching and testing the end condition (control hazards) is less significant. Branch prediction also reduces the number of control hazards. Moreover, unrolling can also reduce the number of data hazards by producing a series of independent instructions that can be reordered by the compiler in the most efficient way. Figure 3.12 is an example that calculates the sum of a group of numbers without unrolling. Figure 3.13 uses loop unrolling. Notice that in this example, temporary variables are used to avoid data dependencies. Table 3.7 shows the performance results.

In the non-optimized code, a dependency exists (at the resultant variable) between two successive iterations. The code cannot be overlapped in the pipeline execution. With branch prediction, this is can have more of an effect than control hazards. In the unrolled code, the loop iteration is replicated four times. In each iteration, the loop updates four sums separately. At the end of the loop, the partial sums are combined. Since the data in the four partial sums are independent of each other, they can be executed simultaneously in the pipeline.

```
double *x;
// Allocate x array

s = 0.0
for (i = 0; i < N; ++i)
  s += x[i];
```

Figure 3.12: Loop unrolling example

```
double *x;
// Allocate x array
// assume N is divisible by 4

s0 = s1 = s2 = s3 = 0.0;

for (i = 0; i < N; i += 4) {
  s0 += x[i];
  s1 += x[i+1];
  s2 += x[i+2];
  s3 += x[i+3];
}

s = s0 + s1 + s2 + s3;
```

Figure 3.13: Optimized loop unrolling example

In the PIPT, most loops were implicitly left for the compiler to unroll. Maximum optimization settings for various compilers were included in the configuration script that enables the compiler to automatically unroll loops. Care was taken to ensure that loop indices were integer variables (i.e., `int`) to ensure that the compiler would be able to detect that a given loop is unrollable (see Section 3.6 about problems with automatic compiler optimizations).

	No optimization		Full optimization	
Size (N)	Simple	Unrolled	Simple	Unrolled
128	0.000158	0.000165	0.000003	0.000001
256	0.001914	0.000202	0.000005	0.000002
512	0.005539	0.000224	0.000010	0.000004
1024	0.012669	0.002056	0.000019	0.000007
2048	0.026698	0.005629	0.000037	0.000013
4096	0.055606	0.012738	0.000074	0.000025
8192	0.112081	0.027401	0.000147	0.000056
16384	0.225119	0.057218	0.000295	0.000111
32768	0.455112	0.115569	0.000590	0.000223
65536	0.907952	0.234818	0.001182	0.000455
131072	1.829500	0.467329	0.002590	0.001186
262144	3.686710	0.935831	0.007833	0.004969
524288	7.362280	1.940970	0.015990	0.010211
1048576	14.859200	3.819780	0.032143	0.020427
2097152	30.173500	7.673090	0.064132	0.040855

Table 3.7: Loop unrolling performance results

3.5.2 Dependencies Elimination

Dependencies in loop iterations can cause data hazards, which introduce stalls into the pipeline. Data dependencies can be eliminated by rearranging code as shown in Figure 3.14, the ith iteration of the loop references element $[i-10]$. The loop is said to have a dependence distance of 10. To increase instruction level parallelism, the loop is unrolled (Figure 3.15). Because the recurrences in the loop do not depend on each other, they can be executed sequentially in the pipeline.

The results show that the code from Figure 3.15 actually hurt performance. Compiler optimization is usually sophisticated enough to remove simple dependencies. The data dependency removal in the example got in the way of the compiler optimization.

```
double *x;
// Allocate x array

for(i = 10; i < N; ++i)
  x[i] = x[i-10] + x[i];
```

Figure 3.14: Dependency example

A code review of the PIPT ensured that all possible dependencies were eliminated from all kernels and

```
double *x;
// Allocate x array

for(i = 10; i < N-9; i += 10) {
  x[i  ] = x[i-10] + x[i  ];
  x[i+1] = x[i-9 ] + x[i+1];
  x[i+2] = x[i-8 ] + x[i+2];
  x[i+3] = x[i-7 ] + x[i+3];
  x[i+4] = x[i-6 ] + x[i+4];
  x[i+5] = x[i-5 ] + x[i+5];
  x[i+6] = x[i-4 ] + x[i+6];
  x[i+7] = x[i-3 ] + x[i+7];
  x[i+8] = x[i-2 ] + x[i+8];
  x[i+9] = x[i-1 ] + x[i+9];
}
```

Figure 3.15: Optimized dependency example

	No optimization		Full optimization	
Size (N)	Simple	No dependency	Simple	No dependency
32	0.000005	0.000003	0.000001	0.000001
64	0.000012	0.000007	0.000002	0.000002
128	0.000025	0.000016	0.000003	0.000003
256	0.000051	0.000034	0.000005	0.000006
512	0.000103	0.000070	0.000009	0.000011
1024	0.000495	0.000264	0.000019	0.000021
2048	0.000796	0.000418	0.000037	0.000043
4096	0.001078	0.000709	0.000074	0.000085
8192	0.001959	0.001361	0.000184	0.000204
16384	0.003735	0.002619	0.000368	0.000408
32768	0.007221	0.005057	0.000738	0.000816
65536	0.014298	0.010029	0.001474	0.001632
131072	0.028565	0.020069	0.003204	0.003543
262144	0.059179	0.042060	0.009088	0.009382
524288	0.118369	0.084747	0.018401	0.020832
1048576	0.236596	0.169488	0.036923	0.041700
2097152	0.473548	0.338979	0.074038	0.083385

Table 3.8: Dependency elimination performance results

routines. Since at least one compiler (KCC) has shown that it tends to hurt performance by unrolling to remove dependencies, most dependencies were simply noted in the code for possible future source code-level optimization.

3.6 Architecture Specific Issues

It is extremely difficult to write portable high performance software for a wide variety of platforms and operating systems. Vendors tend to have different (and sometimes conflicting) models and procedures for the same type of functionality. For example, there are a large number of "similar-but-different" examples between the two major versions of Unix, System V and BSD. The signal interface in C is a common example. Although programs in both flavors of Unix can receive asynchronous signals, the prototype for the user callback function is different. Authors of portable software must include conditional compilation structures such that the "right" pieces of code are compiled on each different system.

Other factors make it difficult to write portable high-performance software, such as different compiler implementations, levels of standards conformance, and differences in underlying hardware.

3.6.1 Compilers

Even though two different compilers can take the same source code and each produce an executable that seems to function identically to the other, the internal structure of the two executables are probably quite different. In particular, optimization options tend to differ significantly between compilers. Native compilers, for example, will attempt to tailor code generation to optimize for speed for their particular operating system and hardware.

However, there are many common techniques which are becoming standard in most compilers, such as loop unrolling, loop invariant extraction, and dead code removal. But even these simple techniques require significant analysis on the part of the compiler, and, more importantly, the compiler writer. Just like with other marketable software, compiler writers have to test and debug their code, include new features, and meet release deadlines. Since compiler features are largely driven by user requests, they are typically prioritized by how many users ask for each feature. As such, the most common and obvious features are included in the compiler, while more subtle or involved features are not. Indeed, a particular feature can be included in a release, but simple derivations of that feature are not.

Examples of this are evident in the Sun Workshop C compiler (version 4.2). This compiler will only unroll loops that have signed integer indices, and have a condition test in the form of "`index < constant`". The compiler will not unroll the loop if, for example, the primary index is an unsigned integer, or a long (signed or unsigned), or if the condition test is not "<".

While changing the compiler to automatically unroll loops in all of the conditions listed above is probably a fairly simple operation, it would probably take a compiler writer at least a day to implement the necessary changes, write the necessary documentation, and fully test the new code. Having the compiler automatically unroll most kinds of loops is arguably a good feature to have, but a programmer can manually unroll loops and achieve the same result as if the compiler had unrolled it. Compiler writers spend their time implementing more difficult and involved features, particularly ones that cannot be worked around by the user.

Therefore, the only way to ensure that common source-code optimizations are performed on all architectures is to perform them manually. In the PIPT, several source code modifications (described in Sections 3.4 and 3.5) were included to ensure that, regardless of the underlying compiler, several types of source code optimizations are always performed.

3.6.2 Standards Compliance

Differences in numerical precision can also hinder portability of scientific code. Two possibilities exist if the same source code generates different numerical answers on different types of machines:

- Problems exist within the hardware, operating system, or compiler, or
- Problems exist within the user code

Experience has shown that user code is usually at fault.

Fortunately, most modern microprocessors implement the 1985 IEEE standard for floating point mathematics [13], meaning that most numerical precision problems are probably the fault of the programmer. This is not to say that bugs affecting numerical precision do not exist; Figure 3.16 shows a code snipit that produces different answers on different machines. Table 3.9 lists several architectures and the results of the code show in Figure 3.16.

```
#include <math.h>

double x = 0.0;
printf("Arc tangent of 0.0 is: %f\n", atan2(x, x));
```

Figure 3.16: Software precision example

In particular, it was discovered that OSF Unix v2.0 and OSF Unix v3.2 running on Dec Alpha hardware did not adhere to the 1985 IEEE standard for floating point mathematics. The PIPT therefore failed its test suite on these two operating systems, because the mathematical rules used to generate some of the results were different than on all other supported architectures. As a result, the PIPT was declared not to be compatible with OSF 2.0 and 3.2 (see Section A.2).

Architecture	Operating System	Compiler	Result
Sun UltraSPARC	Solaris 2.5.1	Workshop v4.2	0.0
IBM SP	AIX 4.1.1	`xlc` v2.0.1.14	0.0
SGI Origin 2000	IRIX 6.4	MIPSpro `cc` v7.2	0.0
DEC Alpha	OSF v4.0	DEC C V5.2-036	0.0
Convex 3820	ConvexOS 11.5	CONVEX CC v5.0	$\frac{\pi}{2}$
HP Apollo	HP-UX 10.20	HP C Compiler A.10.32.03	NaN

Table 3.9: Results of `atan2(0.0, 0.0)` on different systems

3.6.3 Underlying Hardware

Most complex software systems include some architecture-dependent code; differences for underlying hardware must be accounted for in some layer of the software. In order to attain high performance, software *must* be aware of the underlying hardware and operating system. This adds complexity to portable software packages.

For example, the PIPT uses MPI to effect message passing. While there are public domain implementations of MPI that compile on many different architectures, these packages have sophisticated conditional compilation structures that select the right code for the underlying hardware and operating system. Typical public domain MPI implementations include code for socket-based (TCP/IP) message passing as well as POSIX shared memory calls. MPICH, a public domain implementation from Argonne National Laboratory [7], currently includes conditional compilation for twenty-seven different operating systems and eight different message passing models.

Some hardware differences between systems can be prohibitive to group parallel computations. On a DEC Alpha, the size of a `long` in C is eight bytes; on all other PIPT-supported architectures, the size of a `long` is four bytes. Special provisions had to be inserted in the message passing layer in the PIPT to convert from four to eight bytes (and vice-versa) when a DEC Alpha is used in heterogeneous environments. This conversion is slightly slower, but this penalty was judged to be worthwhile since it allows faster Alpha machines to run with potentially slower workstation machines.

3.7 Multithreading Strategies

The current PIPT implementation includes threading on SMP workers. When a SMP worker receives a slice, it divides it into n sub-slices, where n is the number of CPUs in the machine, unless overridden by the user. n threads are created to process the sub-slices. The threads operate on their sub-slice independently of the other threads, and die when they are finished. Figures 2.10 through 2.12 (Section 2.2) show this scheme.

3.7.1 Counting Workers

It would seem natural to classify each CPU in an SMP machine as a worker itself. This would raise the total number of workers, and reduce the amount of work given to non-SMP workers. Every CPU (as opposed to every machine) would then receive an equal amount of work. Table 3.10 lists a sample heterogeneous configuration with two workstations and two SMP's, and compares the percentage of the input image that each worker would receive in both the current PIPT implementation and this proposed scheme.

	Number	Number of workers	
Worker type	CPU's	By machine	By CPU
Workstation	1	1	1
Workstation	1	1	1
SMP	2	1	2
SMP	4	1	4
Total number of workers:		4	8
Slice % of original image:		25%	12.5%

Table 3.10: Comparison of slice size when counting workers by machines available and by CPU's available

When counting workers by the number of machines available, SMP's will finish faster and sit idle while waiting for the workstations to finish (load balancing helps this situation, see Chapter 4). But when counting workers by the numbers of CPU's available, every CPU receives the same amount of work, and all finish at approximately the same time. No worker needs to idle while waiting for another worker to complete its slice.

Unfortunately, giving each CPU its own slice entails significant restructuring of the internal PIPT slice distribution and collection engine. While the analysis of this feature indicates potential speedup from its implementation, it was not the focus of this work, and was not implemented.

3.7.2 Thread Safety

Since a PIPT worker may be running on an SMP, the PIPT uses multiple threads to process a single image slice. As such, the target image processing routine's window operator routine may be invoked multiple times concurrently on a single memory image. This means that the window operator routine must be thread safe, or the results will be undefined.

Thread safety usually entails ensuring that a particular instance of a function either only uses local resources or has exclusive use of global resources. Local resources generally means that automatic variables should be used in the routine, and that the `static` identifier should *not* be used, as this would only create one instance of a variable that is shared across multiple instances of the function.

However, shared local (or global) resources can be used provided that mutual exclusion is ensured. The PIPT provides wrappers to the operating system mutual exclusion locking system (see the manual pages for `PIPT_Mutex` et al. in the PIPT distribution), but care must be taken in designing mutual exclusion algorithms to ensure that the process is not serialized. For example, the code in Figure 3.17 is essentially serialized. Even if multiple threads invoke the `foo()` function more-or-less simultaneously, only one thread is allowed in the critical section at a time. Thus, any gains from parallelizing this function are effectively lost.

```
void foo()
{
  static double bar;
  static PIPT_Mutex mutex;

  PIPT_Mutex_lock(mutex);
  // Some operation on bar
  PIPT_Mutex_unlock(mutex);
}
```

Figure 3.17: Example function that, even if invoked with multiple threads, will still run in serial

Once the multi-threaded workers were implemented, all image processing window operator routines were examined and made thread safe.

3.7.3 Problems With Multithreading

Unfortunately, the only existing thread safe MPI implementation is IBM's native MPI, which only runs on the IBM SP series of machines. Section 4.4 discusses a particular problem related to multithreading and load balancing on SMP's which entails the need in the PIPT for a thread-safe implementation of MPI.

Additionally, the extra memory copy that is avoided on SMP workers by the use of multithreading was problematic on SMP managers. When the manager is an SMP, Section 2.2 discusses how the worker that co-resides on the same machine need not have a slice sent to it via MPI – it can simply use the slice in place (since the manager and worker share the same memory space, the worker can use the slice that already exists in the manager's memory space). However, this scheme requires a thread-safe MPI implementation, which is not freely available. So the slice must be copied to a separate buffer for the worker to process, even though the manager and worker share common memory.

3.8 Slice Distribution

In the PIPT, input images are broken into multiple sections (commonly referred to as "slices"). These slices are distributed by the manager to the workers who perform the image processing computations, and then send their slices back to the manager. Combined with the different load balancing schemes (see Chapter 4), the slicing scheme adapts well into different computing environments, especially active heterogeneous workstation clusters.

Each worker currently receives one slice at a time to process. In view of different compute speeds in a heterogeneous situation, it would be advantageous to have the manager adaptively size the slice that is sent to a given worker based upon the worker's past performance, perhaps as a function of pixels processed per second. While past performance does not predict future performance, this adaptive heuristic paired with the RFFFS load balancing algorithm could yield good results.

However, implementing the differently-size slices would require a different internal model than the PIPT currently implements. Significant portions of the opaque transport mechanism would have to be restructured to support such a scheme.

Sending differently-sized slices to workers can be viewed as sending multiple single-row slices. Unfortunately, taking this approach also entails significant re-coding of the opaque transport mechanism in the PIPT. Therefore, in the current version of the PIPT, each worker still only receives one slice at a time.

Chapter 4

Load Balancing

4.1 Overview

To extract maximum performance and efficiency from a parallel computing environment, it is critical that the workload be properly distributed among the processing nodes. Since the entire parallel computation cannot be completed until all nodes have completed their portion of the overall task, the parallel computation will be limited by the slowest processor. A single processor taking longer to complete its task than the other processors will hold up the entire parallel computation.

In an ideally distributed parallel computing task, all the processing nodes will complete their tasks at precisely the same time. In active heterogeneous workstation clusters, this is difficult to achieve; workstations have different clock speeds, different amounts of RAM, etc. Additionally, the CPU load on a workstation can change frequently during a single parallel run due to irregular user usage patterns.

The goal of load balancing is to partition the parallel task in such a way that each processor will work for approximately the same amount of time, but not necessarily perform the same amount of work. The PIPT incorporates three different load balancing algorithms: no load balancing, simple first-finish, first-serve (FFFS), and redundant first-finish, first-serve (RFFFS). FFFS and RFFFS have additional parameters which can be used to adjust performance for different types of environments. The different algorithms are discussed below.

4.2 No Load Balancing

This algorithm is best suited for use on dedicated parallel hardware. Since all nodes will have the same compute power available for the length of the entire run, distributing the work evenly across the worker nodes makes the most efficient use of computing resources. In this case, the PIPT only divides the work up into as many slices are there are worker nodes; each worker receives one slice to process. The data flow is similar to that shown in Figure 2.8 (page 11). Pseudocode for this algorithm is shown in Figure 4.1.

```
/* Divide the input image up into slices */
$input\_slices \leftarrow$ divide_image($input\_image$, $num\_workers$)

/* Send the slices to the workers */
for $i \leftarrow 1 \ldots num\_workers$
   send($i, input\_slices[i]$)

/* Receive the processed slices back into the output image */
for $i \leftarrow 1 \ldots num\_workers$
   $who \leftarrow$ wait_for_returned_slice()
   receive($who, output\_slices[who]$)
```

Figure 4.1: No load balancing algorithm

Since each node is equivalent, they are each given the same amount of work. It is expected that all workers will finish in approximately the same amount of time.

4.3 First-Finish, First-Served (FFFS)

It is frequently desirable to run in environments where the compute power of each node is not equivalent. Clusters of heterogeneous workstations naturally have different compute speeds. Clusters are also usually shared resources; other users may be running programs on the workstations that the PIPT is running on. As such, the PIPT will be competing for CPU cycles.

The FFFS load balancing algorithm divides the work into more sections than there are nodes available. A heuristic in the PIPT initially divides the work into $3 \times num\_workers$ sections, but this setting can be changed by the user. Each node is initially given a section of the work to process. The faster (or less loaded) nodes will naturally finish their section quickly, return it to the manager, and request more work. The algorithm for FFFS is shown in Figure 4.2.

In this manner, a fast worker may process multiple slices in the time that it takes a slow (or loaded) worker to process one slice. Since the run is limited by the slowest node, the FFFS algorithm gives more work to faster nodes and less work to slower nodes.

```
/* Divide the input image up into slices */
$input\_slices \leftarrow$ divide_image($input\_image$, $num\_slices$)

/* Send the first num_workers slices to the workers */
for $send\_num \leftarrow 1 \ldots num\_workers$
    send($send\_num, input\_slices[send\_num]$)
$send\_num \leftarrow send\_num + 1$

/* Receive the slices back from the workers */
for $i \leftarrow 1 \ldots$ length($input\_slices$)
    ($who, slice\_num$) $\leftarrow$ wait_for_returned_slice()
    receive($who, output\_slices[slice\_num]$)

    /* Send another slice if there are any left */
    if ($send\_num <= num\_slices$)
        send($who, input\_slices[send\_num]$)
        $send\_num \leftarrow send\_num + 1$
```

Figure 4.2: First-Finish, First-Serve load balancing algorithm.

4.4 Redundant First-Finish, First-Served (RFFFS)

FFFS, while it will usually outperform no load balancing in heterogeneous or unevenly loaded situations, can still end up waiting for a slow node at the end of the computation. In some cases, a slow node can be assigned another section of work just before all the other sections are completed. The job must then wait for the slow node to finish its slice, even though the other nodes are idle.

Redundant FFFS is an extension to the original FFFS algorithm. It exploits a special case in the FFFS algorithm that occurs when a worker node returns a slice and all remaining work sections have already been distributed to other workers. In this situation, FFFS lets the worker remain idle until the end of the run. This is wasteful if fast nodes return their slices while a slow nodes are still processing their slices; the fast nodes sit idle waiting for the slow nodes to finish.

Instead of letting the fast worker node sit idle, RFFFS gives the workers another slice of work. The section of work is already being processed by some other worker node (since all remaining work sections have already been distributed). That is, multiple worker nodes are processing the same slice.

In this way, fast worker nodes can be assigned the same work that a slow worker is processing. The fast workers will finish the section first and return it to the manager, thereby ending the computation (without having to wait for the slow node to finish). Pseudocode for the RFFFS algorithm is shown in

Figure 4.3.

There are additional details that are not shown in Figure 4.3; the `find_unfinished_slice()` routine needs to not only find an unfinished slice, but one with the shortest `redundant_workers_list` (i.e., the slice that has the least number of workers already processing it) in order to prevent clustering of redundant work.

The receipt of the abort message requires asynchronous message receives, which can be implemented with multi-threading message passing code in the workers. That is, the worker must simultaneously block on the potential receipt of an abort message (via MPI) and process its slice at the same time. Unfortunately, there are no MPI implementations for workstation clusters that are thread safe, so blocking on the receipt of a message while processing a slice was not possible. As such, the abort signal was not implemented into the current version of the PIPT.

Instead of aborting the worker nodes, the manager places the redundant workers in a "working" state. When the entire input image has been processed, the manager returns the output to the calling function, regardless of whether the redundant nodes have completed their sections or not. The redundant nodes are reclaimed later; when a redundant node finally attempts to return its processed (but now outdated) slice, the manager ignores the slice and tells the worker to join the current computation. These complexities are only required because MPI implementations are not thread safe, and are not shown in Figure 4.3.

4.5 Comparison of Algorithms

The PIPT works well without load balancing on dedicated parallel hardware; there is no need for the additional overhead of load balancing when all the compute nodes are both equivalent in compute power and dedicated to the PIPT job. But in an active heterogeneous workstation cluster, this algorithm performs poorly. Figure 4.4 shows a timing plot of a run with no load balancing on an active workstation cluster. The horizontal lines represent the manager and the workers; the top line is the manager, the bottom four lines are the workers. Vertical lines represent messages between processors. Green areas indicate computations contributing to that worker's slice; red areas indicate blocking in message passing calls.

It is clear from Figure 4.4 that in an active workstation cluster, run times can be very large without load balancing. Figure 4.5 shows the PIPT using FFFS on the same workstation cluster.

The FFFS algorithm shows a marked improvement in performance over no load balancing, but worker 2 still delays the end of the computation. Figure 4.6 shows the PIPT using the RFFFS algorithm on the same workstation cluster. Workers 1 and 4 both perform more work than workers 2 and 3, but also redundantly process the same slices as workers 2 and 3, resulting in a shorter run time for the overall computation.

```
/* Divide the input image up into slices */
$input\_slices \leftarrow$ divide_image($input\_image$, $num\_slices$)

/* Send the first num_workers slices to the workers */
for $send\_num \leftarrow 1 \ldots num\_workers$
  mark_not_done($input\_slices[send\_num]$)
  add_to_list($send\_num, input\_slices[send\_num].redundant\_workers\_list$)
  send($send\_num, input\_slices[send\_num]$)
  mark_as_working($send\_num$)
$send\_num \leftarrow send\_num + 1$

for $i \leftarrow 1 \ldots num\_slices$
  ($who, slice\_num$) $\leftarrow$ wait_for_message()

  /* If the node has a slice to return, receive it, and send
    abort messages to redundant workers */
  if (marked_as_working($who$))
    receive($who, output\_slices[slice\_num]$)
    mark_done($input\_slices[slice\_num]$)
    remove_from_list($who, input\_slices[slice\_num].redundant\_worker\_list$)
    send_abort_messages($input\_slices[slice\_num].redundant\_worker\_list$)
    mark_as_idle($input\_slices[slice\_num].redundant\_worker\_list$)

  /* Send another slice if there are any left */
  if ($send\_num <= num\_slices$)
    add_to_list($who, input\_slices[send\_num].redundant\_workers\_list$)
    send($who, input\_slices[send\_num]$)
    mark_as_working($who$))
    $send\_num \leftarrow send\_num + 1$

  /* If all slices have been distributed, send out a redundant
    slice */
  else if (count_how_many_done($input\_slices$) < $num\_slices$)
    $num \leftarrow$ find_unfinished_slice($input\_slices$)
    add_to_list($who, input\_slices[num].redundant\_workers\_list$)
    send($who, input\_slices[num]$)
    mark_as_working($who$))
```

Figure 4.3: Redundant First-Finish, First-Serve load balancing algorithm

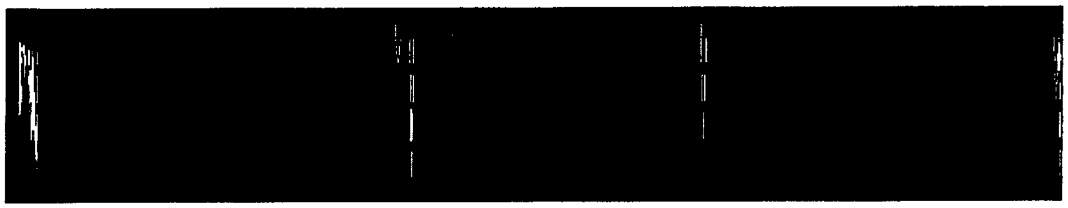


Figure 4.4: A PIPT run with no load balancing on an active workstation cluster. Workers 2 and 3 are heavily loaded, and delay the completion of the computation.

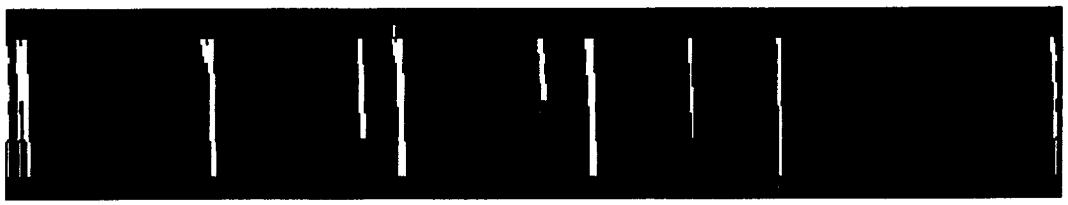


Figure 4.5: A PIPT run using FFFS load balancing on an active workstation cluster. Workers 2 and 3 are heavily loaded, but workers 1 and 4 compensate by processing more slices than workers 2 and 3.

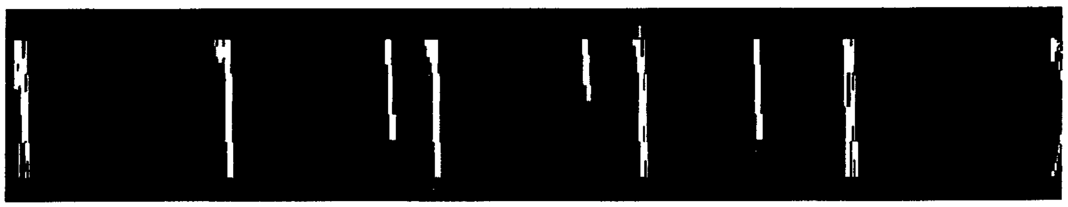


Figure 4.6: A PIPT run using RFFFS load balancing on an active workstation cluster. Workers 2 and 3 are heavily loaded, but workers 1 and 4 compensate by performing more work than workers 2 and 3, and then redundantly processing the final slices of workers 2 and 3. Notice that workers 2 and 3 keep computing even after the manager finishes the computation.

4.6 Communication Costs

The PIPT uses a manager/worker paradigm, in which the manager receives the input image, partitions the work, distributes the image slices to the workers, and finally gathers the processed slices back from the workers. Sending and receiving the image slices across an interconnection network incurs some delay which contributes to the overall wall-clock execution time. Two factors which contribute to network overhead are latency and the speed of the network.

While many workstation networks operate at 10Mbps, faster speeds, such as 100Mbps and 155Mbps, are becoming more common. Naturally, since the PIPT sends large messages across the network, a fast underlying transport will reduce the amount of overhead incurred while processing images in parallel.

4.6.1 Latency

Figure 4.7 illustrates the costs of sending messages using MPI in a workstation cluster environment. Messages of sizes from 1 to 1,048,576 bytes (1 Mbyte) were sent from manager to worker on 10Mbps and 100Mbps switched ethernet networks, as well as a 155Mbps ATM network. Note that the cost for the smaller messages is dominated by latency time (approximately 0.5 milliseconds), but that for larger messages, the cost becomes linearly proportional to the message size. One conclusion that can be drawn is that in workstation cluster-based computation, one should strive to minimize the number of communication operations and at the same time, one should maximize the size of the messages that are sent.

Figure 4.7 shows the peak MPI bandwidths of the 10Mbps, 100Mbps, and 155Mbps networks to be approximately 8.3Mbps, 81.1Mbps, and 113Mbps, respectively. But notice that the latency for 100Mbps is somewhat lower than for 10Mbps or 155Mbps. This is probably due to differences in Solaris networking drivers; while all machines were UltraSPARC 140e's running Solaris 2.5.1, the machines where the 10Mbps tests were run were at a different operating system patch level than the machines that ran the 100Mbps tests. Even though the same machines ran both the ATM and 100Mbps tests, the higher ATM latency can also be explained through operating system network drivers; the ATM driver goes through a TCP/IP stack followed by an internal ATM protocol stack.

4.6.2 Network Speed

The input image is scattered to the workers, and then the output image is gathered back to the manager. Since the output input size is frequently the same as the input image size, the amount of data transmitted across the network is usually twice the size of the input image. For a given message *size* and network *speed*, the theoretical time required for transmission can be calculated by

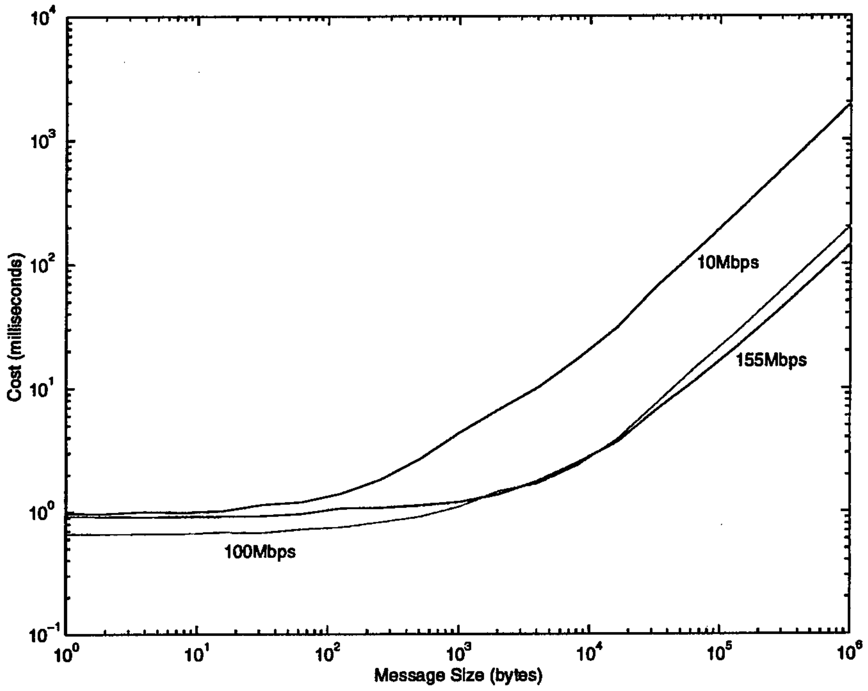


Figure 4.7: Time required to assemble and send messages of different sizes on 10Mbps, 100Mbps, and 155Mbps workstation networks.

$$size \text{ bytes} * \frac{8 \text{ bits}}{1 \text{ byte}} * \frac{1 \text{ second}}{speed \text{ bits}} = \text{transmission time}$$

For example, to send 1,048,576 bytes (1 megabyte) across a 10Mbps network

$$1,048,576 \text{ bytes} * \frac{8 \text{ bits}}{1 \text{ byte}} * \frac{1 \text{ second}}{10,485,760 \text{ bits}} = 0.8 \text{ seconds}$$

But these formulas represent an ideal case; they do not take into account physical processing speeds, network congestion, protocol overhead, etc. Using Sun UltraSPARC workstations running Solaris 2.5.1, timing tests were conducted on 10Mbps and 100Mbps switched ethernet networks, and a 155Mbps ATM network. The public domain software `nttcp` was used to measure sustained bandwidth by sending 2^{19} 8192 byte packets (a total of 4 terabytes, the most that `nttcp` could handle) between two nodes on the same LAN segment. Table 4.1 shows both the time necessary to send the data, and the sustained bandwidth.

Table 4.1 also shows that ethernet networks typically cannot sustain their peak bandwidths. While the ATM network can sustain a bandwidth very close to its peak (147.77Mbps), ATM hardware is still very

Network type	Time	Sustained bandwidth	Percent of peak
10Mbps switched ethernet	36:44	7.24 Mbps	72.2%
100Mbps switched ethernet	3:34	74.86 Mbps	74.9%
155Mbps ATM	2:32	144.77 Mbps	93.4%

Table 4.1: Comparison of sending 2^{19} packets 8192 bytes each (for a total of 4 terabytes) across different types of networks.

expensive compared to ethernet. Using a 7.24Mbps sustained average for a 10Mbps ethernet, we can recalculate the actual transmission time for 1 megabyte

$$1,048,576 \text{ bytes} * \frac{8 \text{ bits}}{1 \text{ byte}} * \frac{1 \text{ second}}{7,591,690 \text{ bits}} = 1.1 \text{ seconds}$$

File	Size	10Mbps	100Mbps	155Mbps
`big.tif`	8,962,934 bytes	18.89	1.83	0.94
`color.tif`	4,608,604 bytes	9.71	0.94	0.49

Table 4.2: Comparison of round-trip transmission times (in seconds) of files across different speed networks using sustained peaks of 7.24Mbps, 74.86Mbps, and 144.77Mbps.

Table 4.2 shows a comparison of round-trip transmission times for two large images, calculated with average peak sustained rates from Table 4.1. The table shows that faster networks can significantly reduce the overhead time necessary for scattering and gathering images. The `big.tif` file takes nearly 19 seconds to transmit (in raw TCP/IP) on a 10Mbps network, but less than half a second on the ATM network. These times will be slightly higher transmitting the same data in MPI, since MPI protocols add some overhead and decrease peak bandwidth (see Figure 4.7). This is important because this transmission time is counted towards the overall wall-clock time for a given PIPT routine. That is, even if the worker nodes are extremely fast machines, any PIPT routine that distributes an input image and gathers an output image processing `big.tif` will take a minimum of 19 seconds on a 10Mbps network due to network transmission time.

4.7 Work Sharing on the Manager

In previous versions of the PIPT, the manager node was dedicated to distributing and gathering image slices from the workers. Slice management tends to occur in bursts, with long periods of inactivity in between (especially with larger images, or when not using load balancing). This is a waste of the manager's compute resources. For example, running the PIPT 1.x with no load balancing on four unloaded workstations on a local area network, only three would actually be performing the image processing calculations.

The PIPT 2.1 now launches a worker on the manager node. In the above mentioned example, all four workstations now contribute to the image processing calculations. This is a much more efficient use of computing resources, since the input image is divided into four slices instead of three; each worker will have over 8% less work to perform.

Chapter 5

Parallel High Resolution Video Stills

5.1 Overview

Accurate image expansion is important in many areas of image analysis. To generate precise maps of the Earth's surface, cartographers must expand small regions of satellite image data. In medical imaging, computerized tomography slices and X-rays may need to be zoomed to search for anomalies. Reconnaissance photographs must be expanded accurately to show hidden details of weapon manufacturing plants and landing strips.

High Resolution of Video Stills (HRVS) addresses how to utilize both the spatial and temporal information present in a short image sequence to create a single high-resolution video frame. HRVS includes two stages: **Motion Detection** and **Bayesian maximum a posteriori (MAP)**[1]. In the motion detection stage, motion vectors between central frame and neighboring frames are computed. In the MAP stage, the motion vectors extracted from neighboring video frames are then used to constrain the MAP algorithm, and a high resolution frame then will be generated.

HRVS shows visual and quantitative improvements over bilinear, cubic B-spline and Bayesian single frame interpolations, which could be used for integrating multiple frames form cockpit or gun video to enhance video frames of interest for exploitation.

Because of the large data set size of the resulting high resolution image data, and the computation-intensive algorithms such as displacement vector detection, the serial implementation of HRVS typically takes hours to finish. Therefore, high-speed implementation of image expansion algorithms is a critical requirement. The high degree of data locality and inherent parallelism in most image expansion algorithms gives the possibility of parallel implementation.

The basic algorithm for single-frame or multi-frame expansion was proposed by Dr. Robert L. Stevenson and Dr. Richard R. Schultz [16, 17]. It uses the Huber-Markov random field (HMRF) model to represent piece-wise smooth data, and displacement vectors are obtained by a hierarchical motion detector.

[1]MAP is a stochastic regularization technique to deal with the ill-posed interpolation problem

In this report, we describe the design of the Parallel HRVS (PHRVS) as a library interface, and we give testing results for both accuracy and performance.

The PHRVS uses the Parallel Image Processing Toolkit (PIPT) [15] as the basic developing tool in order to achieve parallelism. The Visual Instruction Set[2] is used to implement the Mean of Absolute Difference (MAD) to achieve instruction level parallelism.

5.2 Libraries Used in PHRVS

5.2.1 Parallel Image Processing Toolkit

The Parallel Image Processing Toolkit (PIPT) was developed by the Laboratory for Scientific Computing at the University of Notre Dame. It is a scalable, extensible and portable collection of image processing routines, which uses the message passing model based on the Message Passing Interface (MPI) standard, and is specifically designed to support parallel execution on heterogeneous workstation clusters. The PIPT takes advantage of the standard pattern of image/video processing, and hides the parallelism from the library user.

In the latest version of the PIPT (2.1), some new features were added in order to handle the video processing. The most important one that image frames can be registered together, and PIPT treats them as an integrated group and distributes them onto worker nodes in a persistent manner.

The new thread APIs in PIPT-2.1 are also used to add multithreading into PHRVS.

This shows that the PIPT is flexible enough to handle video processing. The programming paradigm used in the PIPT video processing routines is similar to PIPT image processing routines, so a PIPT programmer can write video processing easily using the PIPT-2.1.

5.2.2 Visual Instruction Set

The Visual Instruction Set (VIS) is a set of RISC instructions which are extensions to the SPARC V9 architecture and are designed to accelerate multimedia, image processing, and networking applications. VIS can use four 16-bit adders and four 8x16 multipliers simultaneously. This kind of parallelism (instruction level parallelism) can be used to gain image processing performance without suffering the overhead of data communications.

[2]The Visual Instruction Set (VIS) is a set of RISC instructions which are extensions to the SPARC V9 architecture and are designed to accelerate multimedia, image processing, and networking applications.

5.3 Design

5.3.1 Distributed Memory Parallelism

By using the PIPT, PHRVS can gain parallelism without exposing the details of parallel implementation to user.

In a single frame image expansion routine, the input image is partitioned and distributed by the PIPT kernel to several worker nodes. The resulting expanded sliced images from these worker nodes are finally combined together in the manager node, which then returns one expanded image back to user.

Normal image expansion algorithms can be parallelized easily using this method. However, the MAP expansion algorithm is more difficult to parallelize. This is because the key operation of MAP is to minimize a global function using the method of steepest descent. Thus, one output pixel depends on all the pixels of the input image. This kind of dependency makes parallelism hard to apply. If we simply slice and distribute the image evenly, the resulting image may not match the serial result (this problem is more obvious on the edge pixels of the sliced image). Although this means we will not get the same result in parallel as in serial, we can take advantage of the locality property of the image[3] and apply the MAP method in parallel without suffering too much accuracy loss. This is done by sending some extra overlap rows to the worker nodes. The worker nodes will do some redundant computation on these overlap rows, but more information will be present to expand the sliced image, especially for the edge pixels, and the result will be acceptable. Some experiments we have done show that if six overlap rows are used, the SNR difference between parallel and serial results is less than 0.1, which means no observable difference to a human viewer. Although a large number of overlap rows can be chosen, it is usually not practical, since the redundant computation on overlap regions will hurt the overall performance of PHRVS.

In the video stills expansion routine, the whole process is split into two separate stages. In stage one, the displacement vectors are detected in parallel. In stage two, after the manager gets the whole displacement vector, it redistributes both the image sequences and the corresponding displacement vector field. With the image sequence and displacement vectors, the worker node will use a multi-frame MAP to expand the image slice. The expanded image slices are combined together in the manager node and returned to the user. Again, some overlap rows are added to the slices sent to each worker node.

The reason to separate PHRVS operation into two stages and parallelize them separately is that the motion detection operation fits well into the standard PIPT communication pattern, while the MAP expansion operation has to be treated specially using the method introduced above (by using the overlapping rows).

[3]One output pixel usually depends on the whole input image; the effect of all the pixels in input image on an output pixel, however, can be different. Usually, only a small region of an input image will effect a given output pixel – pixels elsewhere typically have no (or very little) impact on the output image. This is called locality property of image processing.

5.3.2 Instruction Level Parallelism

MAD (mean absolute difference) is an important routine used to estimate block motion between image sequences. The multi-frame MAP algorithm spends nearly 90 percent of its processing time on MAD. MAD is the hotspot for all kinds of block matching algorithms (in our multi-frame MAP, we use the hierarchical sub-pixel motion estimation algorithm to extract the displacement vector between image sequences) since it is used in the inner loop of computation. VIS can be used to implement MAD in order to achieve almost four times speedup.

5.3.3 Shared Memory Parallelism

For the symmetric multiprocessor architecture, we can take advantage of the existing shared memory to avoid the relatively slow network data communications. This will give better performance, and the programming will also be easier than for a distributed system.

Multithreading is a common programming paradigm for multiprocessors, and our PHRVS is multithreaded to take advantage of multiprocessor architecture, if it is available. PIPT offers a multithreaded computing kernel, which means the standard PIPT routines can have multithreading for free. However, PHRVS does not use a standard PIPT computing kernel, thus multithreading had to be done explicitly.

Based on the pipeline-like dataflow property of HRVS, no single busy loop can be located; threading cannot be simply applied to one single loop in order to get better performance. The implementation of Solaris threads is such that the operating system will not allocate a processor for a particular thread until the thread convinces the operating system that it is computationally intensive. This means that creating a thread for each loop would not be effective. In such an implementation, the threads would be very short lived, so they would not have time to move to another processor. As a result, most computation would still be in serial.

The better approach for multithreading PHRVS is to do the thread creation at a higher level. When the multiprocessor machine gets an image slice from the PIPT manager, it will further slice the incoming image into several slices, then create one thread for each slice to do the PHRVS computation. When all the threads finish the computation, all the resulting outputs are collected and combined together to generate the output, and transfer back to the PIPT manager. In short, a **foreman** process is running on each worker node, which further slices the incoming image and spawns threads to work on them in parallel.

The performance data shows that multithreading achieves good speedup on a multiprocessor machine, and is closer to a linear speedup than the distributed system for small numbers of processors.

5.4 Stand-alone application

A PHRVS application is written using PHRVS library routines; it accepts standard command line arguments.

5.5 Library Routines

5.5.1 Parameter Data Structure

`Parms` is a structure used to hold and transfer parameters needed by HRVS routines. Its definition is shown in Figure 5.1.

```
typedef struct {
  int h;      /* height                                 */
  int w;      /* width                                  */
  int q;      /* expansion factor                       */
  FP T;       /* Huber edge penalty function threshold */
  int maxit; /* maximum # of iterations                */
  FP stop;    /* stopping criteria for gradient
                 projection algorithm                   */
  int window_smooth; /* size of smooth window           */
  int window_count;
  FP T_ch;     /* threshold used in UPD                 */
  FP T_ob;
  int border;
  int spacing;
  int p;       /* frame, default = 5                    */
  int d;       /* block, default = 9                    */
  int frames; /* # of frames, default = 3               */
  FP L;        /* lambda = L/|1-k|                      */
  int pan;     /* indicate pan or motion estimation     */
  int interlace; /* indicate PROGRASSIVE or INTERLACED
                    sequence. */
  int field;  /* indicate the first frame of interlaced
                 sequence is even or odd.
                 0 - EVEN, 1 - ODD */
} Parms;
```

Figure 5.1: PHRVS parameter setup

Four routines are defined that allocate and initialize, free, print and copy the Parms structure:

```
Parms* PHRVS_allocParms(int h, int w, int q, FP T, int maxit,
                        FP stop, int window_smooth, int window_count,
                        FP T_ch, FP T_ob, int border, int spacing,
                        int p, int d, int frames, FP L, int pan
                        int interlace, int field);
void PHRVS_freeParms(Parms *p);
void PHRVS_printParms(Parms *p);
void PHRVS_copyParms(Parms *from, Parms *to);
```

5.5.2 Single Frame MAP

One routine is defined to perform a single frame MAP:

```
IMAGE* PHRVS_Pmap(IMAGE *inImage, Parms *parms);
```

The routine takes one input image and the MAP parameters, and returns the expanded image. The parallelism is hidden inside the routine.

5.5.3 Displacement Vector Detector

The following routine is defined to perform displacement vector detection:

```
MotionVectors* PHRVS_Pmotion(IMAGE *inImage[], Parms *parms);
```

It takes an input image sequence and parameters, generates a packed displacement vector structure. A packed D-vector structure can be accepted by the PHRVS routine. It can also be dumped into a file or loaded from a file using the following two routines:

```
void PHRVS_dumpVector(MotionVectors *v, int dim, int h, int w,
                      char *file_name);
MotionVectors* PHRVS_loadVector(char *file_name, int dim, int h,
                                int w);
```

5.5.4 Parallel HRVS

After a displacement vector is obtained, this routine is used to generate the expanded image:

```
IMAGE* PHRVS_Phrvs(IMAGE *inImage[], MotionVectors *packed_v,
                   Parms *parms);
```

Where `inImage` is the input image sequence, `packed_v` is the packed displacement vector which is the output of routine `Pmotion(...)`, and `parms` is the parameter defined by the user.

An example of typical user code is shown in Figure 5.2.

5.6 Test Results

5.6.1 Experimental Setup

All the parallel HRVS tests use an overlap of six rows, and four worker nodes. Unless otherwise noted, experiements were conducted on uniprocessor Sun UltraSPARC 170 workstations.

5.6.2 Airport Sequence

The airport sequence is a synthetic sequence created by translating an aerial view of an airport runway.

Technique	Δ_{SNR} (dB)
Single Frame MAP Estimation, $\alpha = \infty$	1.43
Single Frame MAP Estimation, $\alpha = 1$	1.51
Video Frame Extraction with Motion Estimates, $M = 7, \alpha = \infty, \lambda^{(l,k)} = \frac{10}{\|l-k\|}$	3.47
Video Frame Extraction with Motion Estimates, $M = 7, \alpha = 1, \lambda^{(l,k)} = \frac{10}{\|l-k\|}$	5.48
Video Frame Extraction with Panning, $M = 7, \alpha = \infty, \lambda^{(l,k)} = \frac{1000}{\|l-k\|}$	6.72
Video Frame Extraction with Panning, $M = 7, \alpha = 1, \lambda^{(l,k)} = \frac{1000}{\|l-k\|}$	7.00

Table 5.1: Comparison of Enhancement Methods on the *Airport* Sequence (serial version)

Tables 5.1 and 5.2 show the results of serial and parallel HRVS, where α is the threshold parameter separating the quadratic and linear region defined by Huber edge penalty function. When $\alpha = \infty$, the Huber function becomes a pure quadratic, and the aprior density is a Gauss Markov random field. The Huber function is used to control the likelihood of edges in the image data. l is the ID of the neighboring

```
IMAGE *inImage[];          /* input image sequence */
IMAGE *outImage;           /* output image         */
Parms *parms;              /* parameters           */
MotionVectors *packed_v; /* motion vector        */

/* Initialization */
PIPT_Init(argc, argv);

if (PIPT_errflag) {
    PIPT_perror("PIPT_Init");
    return 1;
  }

/* Load the input images, allocate output image */

/* Allocate and set parameters */
parms = allocParms(h, w, q, T, maxit, stop,
                   window_smooth, window_count, T_ch, T_ob,
                   border, spacing, p, d,
                   frames, L,
                   pan, interlace, field);

/* Motion detection */
MotionVectors* packed_v = Pmotion(inImage, parms);

/* Parallel HRVS */
outImage = Phrvs(inImage, packed_v, parms);

/* Clear up */
freeImage(inImage[]);
freeImage(outImage);
freeParms(parms);

/* Exit */
PIPT_Exit();
```

Figure 5.2: PHRVS user code example

Technique	Δ_{SNR} (dB)
Single Frame MAP Estimation, $\alpha = \infty$	1.41
Single Frame MAP Estimation, $\alpha = 1$	1.50
Video Frame Extraction with Motion Estimates, $M = 7$, $\alpha = \infty$, $\lambda^{(l,k)} = \frac{10}{\lvert l-k \rvert}$	3.38
Video Frame Extraction with Motion Estimates, $M = 7$, $\alpha = 1$, $\lambda^{(l,k)} = \frac{10}{\lvert l-k \rvert}$	5.41
Video Frame Extraction with Panning, $M = 7$, $\alpha = \infty$, $\lambda^{(l,k)} = \frac{1000}{\lvert l-k \rvert}$	6.73
Video Frame Extraction with Panning, $M = 7$, $\alpha = 1$, $\lambda^{(l,k)} = \frac{1000}{\lvert l-k \rvert}$	7.04

Table 5.2: Comparison of Enhancement Methods on the *Airport* Sequence (parallel version)

frame, and k is the ID of the central frame. λ is the weight used to specify the importance of motion vectors (as the distance between frame l and frame k increases, the weight of the corresponding motion vectors will decrease, which will generate less effect on the output image). M indicates the number of frames from the sequence that were used to generate the high resolution still.

Tables 5.1 and 5.2 show that the parallel version is more accurate than the original serial implementation. This is because the "serial" algorithm within the parallel version was improved, it uses a different method, for example, some expressions are evaluated in different order, which will give better accuracy. The difference in accuracy does not come from the parallelism.

The results are also shown in Figures 5.3 and 5.4.

Note that two different random fields are applied in this example, one is Huber-Markov random field (HRMF) and the other is Guassian-Markov random field (GRMF), refer to [16, 17] for more detail on this.

Two different motion detection methods are applied here, one is **pan**, which averages all the motion vectors into one motion vector, the other one just uses the motion vectors without taking the average of it.

5.6.3 Mobile Calendar Sequence

The mobile calendar sequence is real video sequence from the MPEG-1 test set. It shows several objects moving with independant motion in a single scene.

Tables 5.3 and 5.4 show that the parallel version of HRVS does not work as well as serial version in this case, but the accuracy loss of SNR is less than 0.1. This is due to the fact that the parallel version uses an approximation to the iterative algorithm in order to simplify the parallel domain decomposition.

The results are also shown in Figures 5.5 and 5.6.

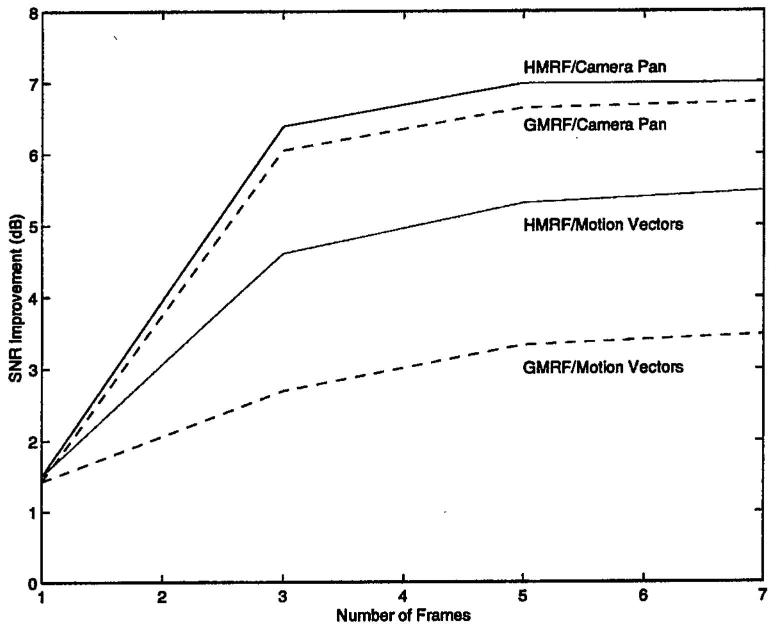


Figure 5.3: Improved SNR versus number of frames (serial version)

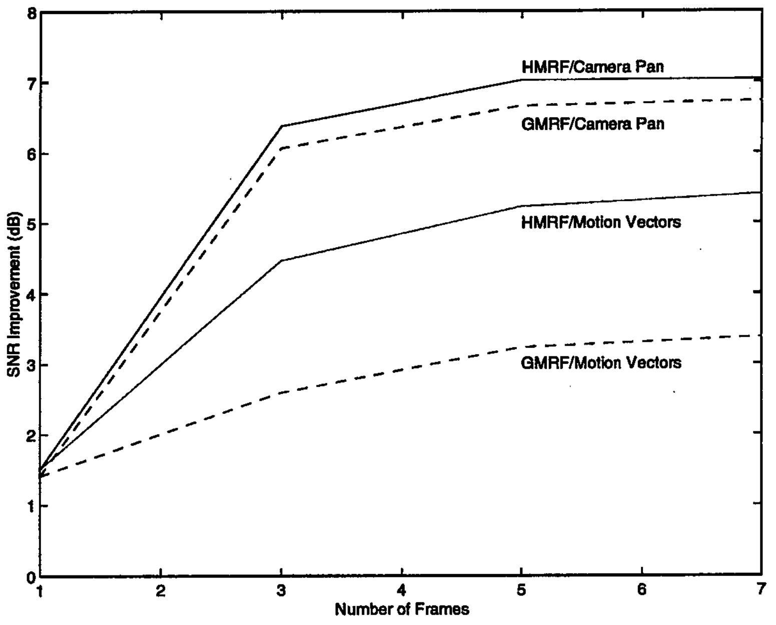


Figure 5.4: Improved SNR versus number of frames in video observation model (parallel version)

Technique	Δ_{SNR} (dB)
Single Frame MAP Estimation, $\alpha = \infty$	0.82
Single Frame MAP Estimation, $\alpha = 1$	1.05
Video Frame Extraction with Motion Estimates, $M = 7$, $\alpha = \infty$, $\lambda^{(l,k)} = \frac{10}{\lvert l-k \rvert}$	1.27
Video Frame Extraction with Motion Estimates, $M = 7$, $\alpha = 1$, $\lambda^{(l,k)} = \frac{10}{\lvert l-k \rvert}$	1.97

Table 5.3: Comparison of Enhancement Methods on the *Mobile Calendar* Sequence (serial version)

Technique	Δ_{SNR} (dB)
Single Frame MAP Estimation, $\alpha = \infty$	1.13
Single Frame MAP Estimation, $\alpha = 1$	1.52
Video Frame Extraction with Motion Estimates, $M = 7$, $\alpha = \infty$, $\lambda^{(l,k)} = \frac{10}{\lvert l-k \rvert}$	1.97
Video Frame Extraction with Motion Estimates, $M = 7$, $\alpha = 1$, $\lambda^{(l,k)} = \frac{10}{\lvert l-k \rvert}$	2.12

Table 5.4: Comparison of Enhancement Methods on the *Mobile Calendar* Sequence (parallel version)

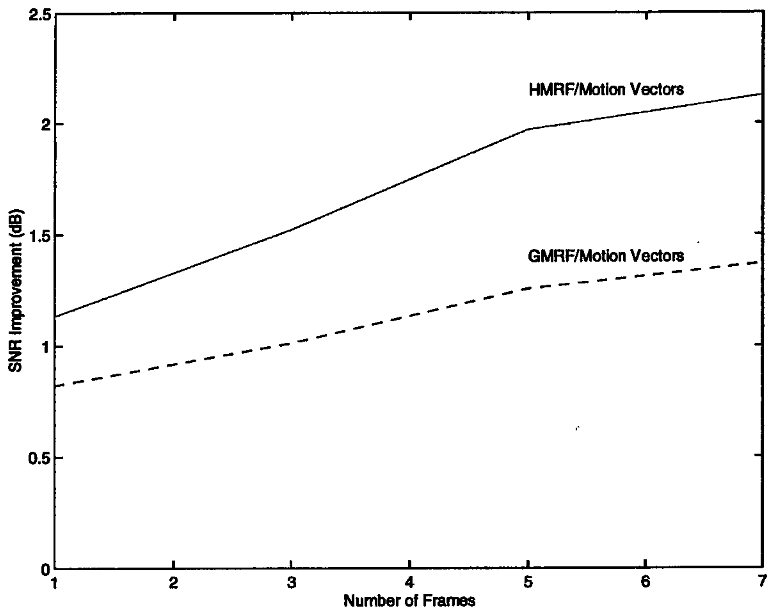


Figure 5.5: Improved SNR versus number of frames in video observation model (serial version)

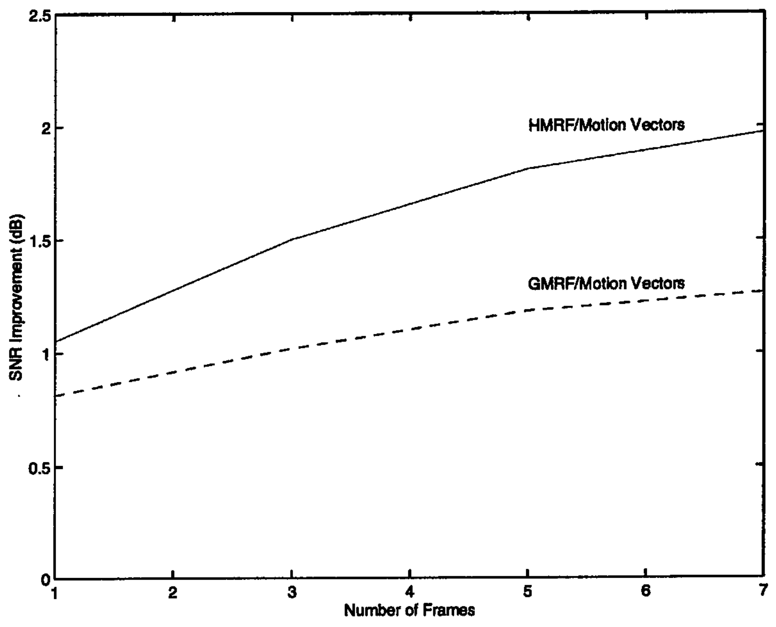


Figure 5.6: Improved SNR versus number of frames (parallel version)

With the result of this test, we can say that the parallel version can be used to gain high performance without sacrificing much accuracy.

Note all the parallel test use an overlap of six rows.

5.6.4 Performance

Performance Gain by VIS

MAD (mean absolute difference) is an important routine used to do motion detection. Because of its standard computational pattern, it can be sped up using VIS. The speedup of the VIS version of MAD versus the normal MAD implementation is shown in Figure 5.7.

The x-axis denotes the size of the blocks operated on, and the y-axis denotes the timing of each MAD implementation.

There are two MAD implementations using normal instructions, and three MAD implementations using VIS instructions. The testing result shows that VIS version of MAD is faster than the normal version of MAD, for some special case (the block size if multiple of 8), a speedup of 4 can be nearly achieved.

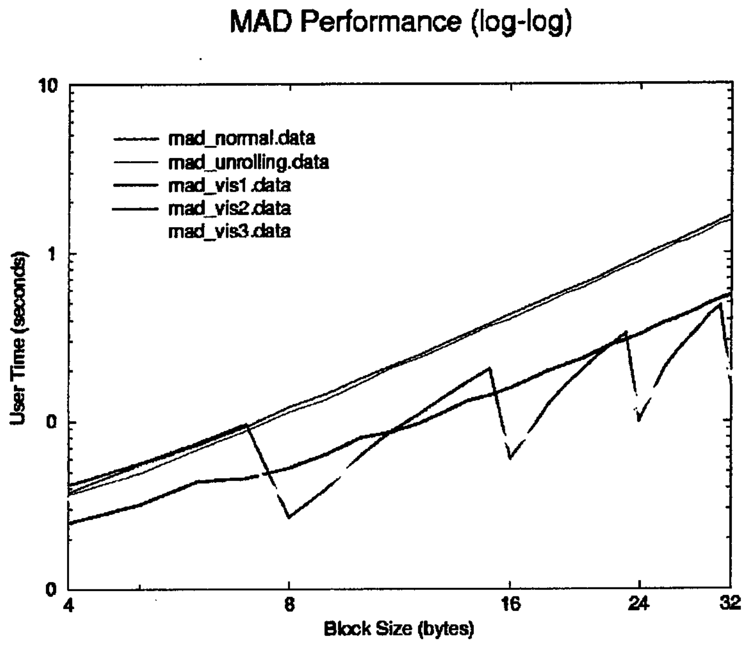


Figure 5.7: Performance of five different MAD implementations

To show the performance of VIS in a real PHRVS application, a three frame airport sequence (64x64) is used as a test case, and the middle frame is expanded by a factor of 4. The wall clock time for PHRVS without VIS is 1426.5 seconds, while the wall clock time for PHRVS with VIS is 383.0 seconds. The speedup is 3.72. It is close to the ideal linear speedup of 4.

Parallel Performance Gains

Since the main goal of parallel HRVS is to get high performance, the speed up of parallel implementation is also tested. The parameters used in testing are shown in Table 5.5. The VIS enhancements were not used in this test.

The overlap is six rows. When PHRVS is applied to the airport video sequence(64x64), we get the performance result shown in Table 5.6. The speedup is not the ideal 4, because the load on worker nodes is not even due to the use of overlap (The middle slices are larger than the edge slices).

The computation time ratio (motion detection vs. MAP) is about 9:1, then $\frac{9}{10}$ of the load is computation of motion vectors, and $\frac{1}{10}$ of the load is on MAP. Suppose we use overlap of 6 rows on 64x64 image expansion, thus the maximum speedup of MAP computation is $\frac{16}{16+6+6} \times 4 = 2.28$ (to compute each slice, 2 overlaps of 6 pixels are needed, thus 28 rows of input pixels are used to compute 16 rows of output pixels); the maximum speedup of motion vectors computation is $\frac{64}{64+6} \times 4 = 3.66$ (), so the total speedup can be expressed as $\frac{1}{\frac{0.9}{3.66}+\frac{0.1}{2.28}} = 3.44$. This approximately matches the performance result

h	=	64
w	=	64
q	=	4
T	=	1.0
maxit	=	50
stop	=	1e-4
window_smooth	=	10
window_count	=	5
T_ch	=	5.0
T_ob	=	10.0
border	=	0
spacing	=	1
p	=	5
d	=	9
frames	=	0
L	=	10.0
pan	=	0

Table 5.5: Parameters setup

Number of frames	Serial	4 processors	Speedup
1	42.106	16.966	2.48
3	381.711	130.889	2.92
5	648.130	209.158	3.10
7	908.794	294.118	3.09

Table 5.6: Performance results

in Table 5.6. Also other overhead such as packing and unpacking motion vectors, redundant MAP computation on the overlap pixels while doing motion detection, and communication overhead has to be considered when comparing the ideal and actual speedups.

The unbalanced load is shown in Figure 5.8.

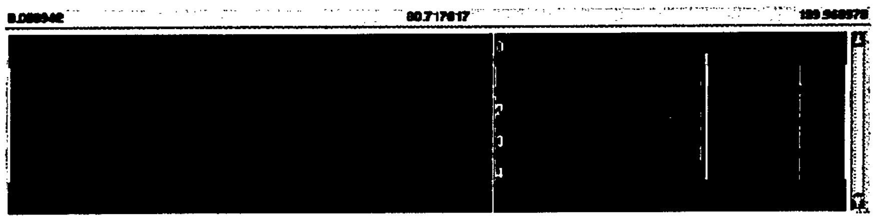


Figure 5.8: Trace for 3-frame hrmf/motion vector (Airport Sequence)

Note the middle two processors get more load than the top and bottom processors. This is the result of giving overlaps of different sizes to the worker nodes.

The same test was done on the mobile calendar sequence (144x180); the results are shown in Table 5.7.

Frames	Serial	2 processors	Speedup	4 processors	Speedup	8 processors	Speedup
1	240.044	132.835	1.81	79.842	3.01	49.688	4.83
3	2348.030	1364.863	1.72	658.584	3.57	414.553	5.66
5	4211.640	2274.864	1.86	1195.730	3.52	674.809	6.24
7	6006.364	3401.058	1.77	1660.635	3.62	1027.015	5.85

Table 5.7: Performance results (Mobile Calendar Sequence)

Using the same computation time ration and overlap as the previous example. Suppose we use an overlap of six rows on a 144x180 image expansion, thus the maximum speedup of MAP computation is $\frac{36}{36+6+6} \times 4 = 3.00$; the maximum speedup of motion vectors computation is $\frac{144}{144+6} \times 4 = 3.84$, so the total speedup can be expressed as $speedup = \frac{1}{\frac{0.9}{3.84} + \frac{0.1}{3.00}} = 3.74$. It closely matches the testing result that we obtained previously (3.62) .

The unbalanced load is also shown Figures 5.9 and 5.10.

Note the white vertical line denotes the data transferring between the nodes.

Comparing Figure 5.8 with Figure 5.9, the latter is more well-balanced than the former. This result is expected by the previous analysis, since for a larger image, the ratio between slice size and overlapped area size is smaller. The ideal MAP parallel speedup will be higher, thus the better result is shown with the larger image.

For Figure 5.10, although the ratio of overlap to window size is the same as the case shown in Fig-

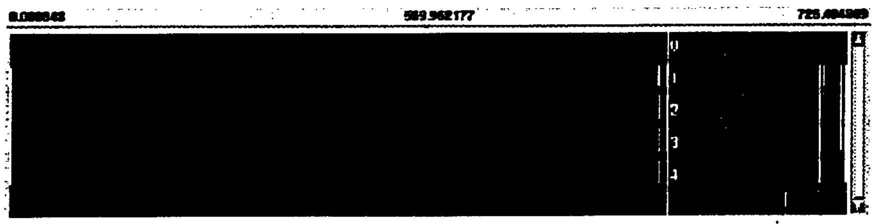


Figure 5.9: Trace for 3-frame hrmf/motion vector (Mobile Calendar Sequence)

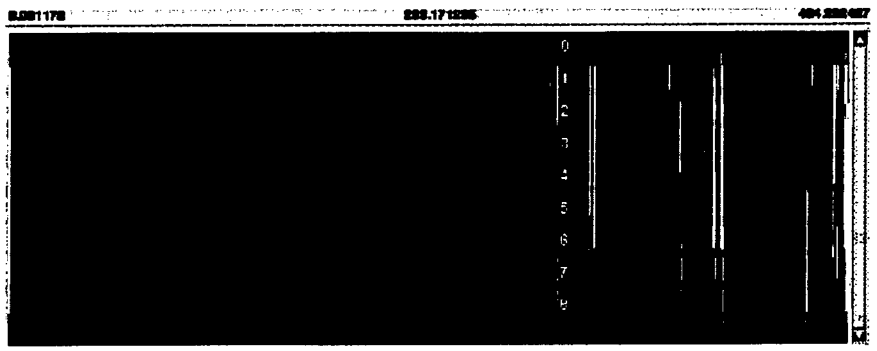


Figure 5.10: Trace for 3-frame Mobile Calendar sequence PHRVS processing with 8 workers.

ure 5.9, more worker nodes are used, and the size of distributed slice image is smaller than the case in in Figure 5.9. Therefore, more redundant computation is performed on each worker node.

5.7 Conclusion

HRVS is a new technology to do image enhancement based on both spatial and temporal information, which gives better visual and quantitative improvements over traditional image enhancement technology. HRVS is a CPU bound job which can benefit a lot by applying parallelization. This is the main motivation of PHRVS.

Both instruction level parallelism and distributed memory parallelism are used in PHRVS.

VIS is the tool used in PHRVS to exploit instruction level parallelism. In the motion detection stage of HRVS, 4 pairs of pixels can be compared in parallel using VIS. The experiment on the airport sequence shows a speedup of 3.72 when using VIS.

PIPT is the tool used in PHRVS to exploit distributed memory parallelism. Video sequences are sliced into several smaller sequences. These sequences are then distributed by PIPT to several working nodes, on which motion detection and MAP computation are done in parallel (each separately).

By applying both instruction level parallelism and distributed memory parallelism, the experiment on 3 airport frames shows that PHRVS gives a speedup of 10.97 over the original serial HRVS.

The work shows the potential of doing HRVS in parallel.

Chapter 6

Parallel Visualization

6.1 Overview

PVIZ (Parallel Visualization Toolkit) is an auxiliary library for the PIPT. The main goal of PVIZ is to offer PIPT users a convenient and efficient way to visualize image processing results, and to perform image processing operations interactively. PVIZ can also be used to build an interface for PIPT applications.

PVIZ is built using Sun's XIL foundation image processing library and the PIPT. It acts as an interpreter between XIL and PIPT, so the user can take advantage of the features in both the PIPT and XIL.

6.2 Supporting Libraries

The PVIZ uses the XIL foundation library to do the basic image processing, while offering the user a PIPT-like interface. Thus, the use of XIL is totally hidden from the PVIZ user interface. The user operates on PIPT `IMAGE` and `PVIZ_Display` objects, and is not required to have any knowledge of XIL foundation library.

6.2.1 Parallel Image Processing Toolkit

The Parallel Image Processing Toolkit (PIPT) [15, 20] was developed by the Laboratory for Scientific Computing at the University of Notre Dame. It is a scalable, extensible, and portable collection of image processing routines, which use a message passing model based on the Message Passing Interface (MPI) standard. It is specifically designed to support parallel execution on heterogeneous workstation clusters.

6.2.2 XIL Foundation Library

XIL is a C language foundation library for image processing and compression. It was introduced in 1992 as a fundamental component of the SunSoft, Inc. Solaris operating system.

As an imaging foundation library, XIL has many good features. The features that are of most interest to us are:

- XIL is MT-hot, meaning that it is both multi-threaded aware and maximizes concurrent processing through the use of threads. Thus visualization will derive a speedup in multiprocessor systems.
- XIL supports deferred execution mode of operation in which execution of a sequence of functions is deferred until certain processing conditions are met. Thus, it accelerates many image processing applications.
- XIL can be used to operate on the frame buffer directly for the purpose of visualization. This saves the extra memory copy and delivers high performance.

For more information about XIL, see [22, 23].

6.3 Design

6.3.1 Image Structure Conversion

The PIPT and XIL use different structures to represent an image. For example, in the PIPT, the image structure is defined in Figure 6.1. The XIL image is an opaque structure.

The XIL image structure has more attributes than the PIPT image, but does share attributes such as height, width, plane, and pixels. In addition to these attributes, an XIL image also contains a storage scheme, a color space, and an origin.

Since the main goal of PVIZ is to use XIL while hiding the internal data representations of XIL, these two image representations have to be converted back and forth. This is done inside the PVIZ routines using (`PVIZ_pipt_to_xil()` and `PVIZ_xil_to_pipt()`). The PVIZ user should not need to call these functions.

Using the conversion routines, each image processing routine in XIL can be wrapped to offer a PIPT-like interface.

```
typedef struct {
  int     fmt;
  uint32  h;        /* height                                */
  uint32  w;        /* width                                 */
  uint32  p;        /* # of planes, 3 for RGB, 1 for Grayscale */
  PIXEL   ***data;  /* raster                                */
                    /* data[0][0] points to the beginning of a */
                    /* contiguous raster of h*w pixels;     */
                    /* data[k][0] points to the beginning of  */
                    /* the k-th plane (or color);           */
                    /* data[k][i] points to the first pixel in */
                    /* the i-th row of the k-th plane       */
  PALETTE *palette; /* Only used for palettized color images  */
} IMAGE;
```

Figure 6.1: The PIPT IMAGE structure.

6.3.2 PVIZ Display

In PVIZ, a display class is supported to display images. Its definition is shown in Figure 6.2.

```
typedef struct {
  Window xwindow;
  XilImage display;
  XilImage buffer; /* double buffer */
  PVIZ_Bool resizable;
  PVIZ_Bool redrawable;
} PVIZ_Display;
```

Figure 6.2: The PVIZ `PVIZ_Display` structure.

There are several operations for manipulating and accessing a display object. These operations include `create()`,`destroy()`,`setDrawable()`,`setResizable()`,and `resize()`. One X window is associated to any created display object. The X window can be retrieved from the display object by calling `PVIZ_Display_getWindow()`. This X window handle can be used in the user interface program.

When a display object is set to be resizable, the size of window will be changed responding to the requirement of different image processing operations. Otherwise, the size of the window will remain the same and the output image may be cropped.

When a display object is set to be redrawable, a double buffer is used to store the intermediate result upon a request or an exposure event, allowing the display to be refreshed automatically. This functionality necessitates some extra overhead from buffering; if it is not necessary, the display should not be set to be redrawable.

6.3.3 PVIZ Routine Interface

Any PVIZ image processing routine can use either a PIPT image object or a PVIZ display object as input and output; there are four interfaces for each PVIZ image processing routine. The last two letters of the function name indicate the type of the input and output arguments. The letter `I` is used indicate PIPT `IMAGE` arguments; `D` is used to indicate `PVIZ_Display` arguments. For example, Table 6.1 shows the four interfaces to `PVIZ_Rotate()`.

Function name	Input type	Output type
`PVIZ_RotateII()`	`IMAGE`	`IMAGE`
`PVIZ_RotateID()`	`IMAGE`	`PVIZ_Display`
`PVIZ_RotateDI()`	`PVIZ_Display`	`IMAGE`
`PVIZ_RotateDD()`	`PVIZ_Display`	`PVIZ_Display`

Table 6.1: The four interfaces to the `PVIZ_Rotate()` function.

Thus, the user can choose to apply an image processing operation in main memory or in the frame buffer. This kind of flexibility is one of the major design goals of PVIZ, and chosen to provide a close mapping to the capabilities offered by XIL.

6.3.4 PVIZ Computation Kernels

New XIL routines can easily be wrapped into PVIZ. To achieve this goal, a kernel registration scheme is used. Each wrapped XIL routine only needs to supply three registration structures that describe what operation it performs and what parameters it needs. After a kernel is registered, all the detailed conversion operations are hidden from the user.

6.4 Demo Program

A simple demo program was written to show how to use PVIZ to write a visualization program. In this demo program, PVIZ is initialized using `PVIZ_Init()`. Two PVIZ displays are then created; the input TIFF formatted image is shown on the main display while the second display is left blank. The user can apply several different image processing operations such as rotation, projection, and edge detection in either the frame buffer or main memory. Help is available by typing '?' in the main window. Users can choose one display as input and the other display as output, or they can choose one display as input and main memory as the output to get a PIPT `IMAGE`. The PIPT `IMAGE` can be also saved as a file or processed using PIPT routines.

The demo interface was written using the X library `Xlib`. User actions are obtained by catching native X events. Since XIL is not tied to any particular X library, users can also use the Motif libraries, for example, to build visualization program interfaces.

6.5 Library Routines

A complete list of the image visualization routines is shown in Table 6.2. Each routine listed has all four interfaces. Refer to [23] for more detailed description of these routines.

Visualization routine	Description
`PVIZ_Combine`	Combine corresponding band of two images into a target image
`PVIZ_Convolve`	Apply convolution operation to an image
`PVIZ_CopyImage`	Copy image
`PVIZ_Dilate`	Apply dilation operation to an image
`PVIZ_Divide`	Apply division operation to two images
`PVIZ_Divide_into_const`	Divide each pixel of image by a constant
`PVIZ_Divide_with_const`	Divide each pixel of image with a constant
`PVIZ_Edge_detection`	Detect the edge of image
`PVIZ_Erode`	Apply erosion operation to an image
`PVIZ_Rotate`	Rotate image
`PVIZ_Rescale`	Rescale each pixel value of image by a certain rate
`PVIZ_Translate`	Move the origin of image
`PVIZ_Transpose`	Transpose an image

Table 6.2: PVIZ image processing library routines.

Chapter 7

Photoshop Interface

7.1 Introduction

In the field of Image Processing, not all users are familiar with programming. In order to provide users with a fast and convenient way to process images, we developed the PIPT [15, 20] plug-in. Plug-ins are software programs that extend the capabilities of existing software. The PIPT plug-in is based on the filter Plug-In of Adobe Photoshop. The PIPT plug-in allows PIPT functionality to become part of Photoshop. It hides the parallelism and programming from the user and provides an effective, stable user visualization environment. In this documentation, we describe the PIPT plug-in, and how this allows the PIPT to be inserted into Adobe Photoshop.

7.2 Requirements

To meet the user's requirements, the PIPT plug-in should meet the following design specifications:

1. **Fast:** Users often require image processing tools to work on very large images, which makes the speed of the image processing critical. To make matters worse, processing time increases polynomially (`O(nm)` for `n x m` image) as the size of the image increases. The PIPT is a parallel image processing toolkit developed by LSC, University of Notre Dame. It uses the Message Passing Interface (MPI) [24] to parallelize the image processing based on the workstation clusters to achieve a high speedup. Therefore, the PIPT plug-in can meet the speed requirement through the use of parallelization in the PIPT.

2. **Popular:** Users have many choices when it comes to software. How can we make the PIPT plug-in a popular choice? Adobe Photoshop is a very popular image processing tool and widely used at the present, so inserting the PIPT into Photoshop can give the PIPT a wide chance to be used. We selected Photoshop as the user interface for the PIPT.

3. **Intuitive GUI:** The PIPT plug-in should have a user-friendly interface. We use the X/Motif libraries to create the graphical user interface for the PIPT plug-in. The user interface is designed to provide a convenient way for users to select PIPT routines, input the parameters and process the image.

7.3 Design

Before introducing the design of the PIPT plug-in, we will talk about how the Photoshop plug-in and PIPT work.

7.3.1 How The Adobe Photoshop Plug-in Works

Plug-in modules provided by Photoshop can allow either the Adobe Systems or third party developers to extend Photoshop without actually modifying the base application. Adobe Photoshop version 3.0 for UNIX supports five kinds of plug-in modules, Acquisition modules, Export modules, Filter modules, File format modules and Extension modules [2].

The Filter module lets you apply special effects by modifying a selected area of an existing image. This is similar in function to the PIPT routines. We use the filter module to realize the PIPT plug-in.

Photoshop for the UNIX platform treats plug-ins as dynamically loaded shared objects. In order to create a shared object when compiling the plug-in program, the correct flag must be present in the makefile, such as in the Sun environment, `-G -Bsymbolic` is used in building the plug-in executable files. Each filter plug-in in Photoshop is required to have an executable file and a resource fork, with an associated file that begins with a percent sign (`%`). The resource fork is used to organize its menus by Photoshop. It is created using the MakePlugIn utility [2]. The executable file is the executable code including the GUI and filter process.

The GUI of a Photoshop Plug-in should be written to conform to Motif style guidelines. To simplify plug-in development, a set of utilities are included in the Photoshop SDK (Software Development Kit). The files `PIUtilities.c` and `PIUtilities.h` contain various routines and macros to make it easier to use the host callback functions.

The program structure of filter plug-ins should obey the following rules:

- The plug-in source file should include the corresponding header file `PIFilter.h`.
- The plug-in must provide a function named `main_entry()`, it is the callback function exported by Photoshop. This is explained later.

- In order to display the plug-in on the screen, we obtain the X display handle through the `PSGet-Display()` routine, or obtain a top level shell using `PSGetTopShell()`. Then we are free to create our own user interface in X/Motif. Photoshop also provides an event dispatching routine, `PSDispatchEvent(XEvent *)` which does Photoshop specific things as well as normal X event processing. It should be (and is) used instead of `XtDispatchEvent()`.

When the user takes an action that causes a plug-in module to be called, Photoshop opens the resource fork of the file that the plug-in module resides in, loads the resource in memory, locks it and calls the routine starting at the first byte of the resource. This must be the `main_entry()` callback function. It has the following calling convention [1]:

```
void main_entry (short selector, Ptr stuff, long *data, short *result);
```

The parameters are to be interpreted as follows:

- `selector`

  This is an integer operation selector code. There are a total of five kinds of selectors defined by filter plug-in. When the user invokes the plug-in by selecting its name from the `Filter` submenu, Photoshop loads the plug-in's resource into memory and calls it with the following sequence of selector values.

  - `filterSelectorParameters`

    If the plug-in filter has any parameters that the user can set, it should prompt the user and save the parameters in a relocatable memory block whose handle is stored in the parameters field. Photoshop initializes the parameters field to NULL when starting up.
  - `filterSelectorPrepare`

    If the plug-in filter needs to allocate large buffers (32K), we set the `bufferSpace` field in `stuff` to the number of bytes it needs. Photoshop will then try to free up that amount of memory before calling the plug-in's `filterSelectorStart` routine.
  - `filterSelectorStart`

    In this call, the fields `inRect` and `outRect` are set to request the first areas of the processed image to work on.
  - `filterSelectorContinue`

    The routine is repeatedly called as long as at least one of the `inRect`, `outRect`, or `maskRect` fields is non-empty.

    It should process the data pointed by `inData` and `outData` and then update `inRect` and `outRect` to request the next area of the image to process.

- `filterSelectorFinish`

  This call allows the plug-in to clean up after a filtering operation. It is made if and only if the `filterSelectorStart` routine returns without error. If Photoshop detects a command-period between calls to the `filterSelectorContinue` routine, it will call the `filterSelectorFinish` routine.

- `stuff`

  This is a pointer to the parameter block. The exact nature of the parameter block depends on the type of plug-in. In filter plug-in, it uses the `FilterRecord` data structure and passes the relevant information of the processed image into `main_entry()`. The `FilterRecord` definition is in the `PIFilter.h` header of the Adobe Plug-In SDK [1].

- `data`

  This is a pointer to a long integer (32 bits) which Photoshop will maintain for the plug-in across invocations. It will be zero the first time the plug-in is called. In the PIPT plug-in, we use this field to store a pointer to the plug-in's "global" data. In the PIPT, we designed the specific global data structure.

- `result`

  This is a pointer to the result code to be returned by the plug-in. It has a different meaning for different values

  $= 0$: no error has occured.

  > 0: The plug-in has already displayed any appropriate error message.

  < 0: The execution of the plug-in should stop, but the host should display its standard error dialog describing the error.

The typical structure of the callback `main_entry()` function is shown in Figure 7.1. All calls to the plug-in module come through this routine. It must be placed first in the resource. To achieve this, the development system requires that this be the first routine in the source code.

7.3.2 Design of the PIPT Photoshop Plug-in

In this section, we will talk about the design of the PIPT plug-in: the main program structure, the graphical user interface and the data structure. The PIPT plug-in is written in the C programming language.

```
extern void
main_entry (short selector, FilterRecord *stuff,
            long *data, short *result)
{
  GHdl globals;         /* defined global data */

  if (!*data) {         /* initialize the global data when the */
                        /* plug-in is first called             */
    *data = (long) NewHandle (sizeof (Globals));
    if (!*data) {
      *result = memFullErr;
      return;
    }
    InitGlobals ((GHdl) *data);
  }

  globals = (GHdl) *data;
  gStuff = stuff;
  gResult = noErr;

  switch (selector) {       /* do the operation for each selector */
    case filterSelectorAbout:    /* display the about dialog box */
      DoAbout (globals);
      break;
    case filterSelectorParameters:
                        /* input the parameters needed by plug-in */
      DoParameters (globals);
      break;
    case filterSelectorPrepare:     /* allocate the large buffer */
      DoPrepare (globals);
      break;
    case filterSelectorStart:       /* set the first processed */
      DoInitialRect(globals);       /* rectangle image */
      break;
    case filterSelectorContinue:    /* process the image */
      DoContinue (globals);
      break;
    case filterSelectorFinish:      /* free the allocated buffer */
      DoFinish (globals);
      break;
    default:
      gResult = filterBadParameters;
      break;
  }

  *result = gResult;    /* return result */
}
```

Figure 7.1: Example of main_entry() function

Program Structure – The Interface With Photoshop

In Photoshop, the PIPT plug-in is under the `Filter` menu. A submenu called "PIPT" is inserted in the `Filter` menu. When the PIPT plug-in is selected, the PIPT plug-in dialog box (see Figure 7.2) appears. A user can choose the PIPT routine to process an image. The PIPT routines are split into five categories - Enhancement, Feature, Filter, Imagehandling, Random. In the PIPT dialog box, a user can also decide how many processors to use in the parallel processing and whether to turn on the timer which reports the processing time. If the user turns on the timer, after the image processing finishes, the PIPT plug-in will display the image processing time.

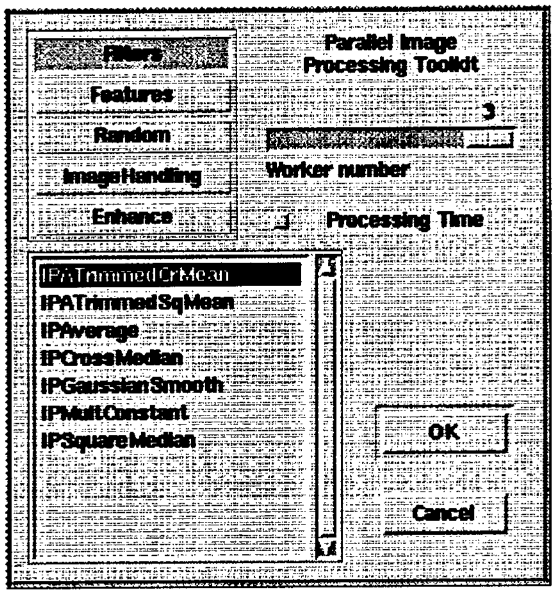


Figure 7.2: PIPT dialog box

Since it is the best choice to insert the PIPT as one submenu into Photoshop `Filter` menu, only one main module `PIPT.c` which consists of the plug-in callback function `main_entry()` is designed according to the rules of Photoshop filter plug-in. In this module, the user selection is dispatched and the parameters input by the user are passed into the PIPT routine.

In Photoshop filter plug-ins, the image is typically split into many slices and passed one at a time to the function selector `filterSelectorContinue`. This is a sequential process until all the slices are processed. In order to make this parallel in the PIPT, we don't split the image. From the point of Photoshop, the whole image is partitioned into only one slice. To achieve this, two steps needs to be done. One is that we set the initial processed slice as the whole image in the function of the selector `filterSelectorStart`, the other is that we should set `bufferSpace` to tell Photoshop the size of the needed buffer in the function of the selector `filterSelectorPrepare`. Because the image is not split, a large buffer is required to store the whole image.

In the program structure shown above, The function `PIPT_UI()` is designed for the PIPT plug-in main dialog box. It will be introduced in Section 7.3.2.

`Dispatch_UI()` brings up the relevant window for a particular PIPT routine according to the user selection. Figure 7.3 shows the parameter input window for the `IPAddUniform()` routine. `Dispatch_UI()` has the following prototype:

```
int Dispatch_UI(GHdl globals);
```

The argument `globals` has the data structure of `GHdl` defined in `PIPT.h`. It stores the global information related to the image processing such as the selected PIPT routine, the number of workers etc. We will introduce the `GHdl` data structure in Section 7.3.2. Although the Photoshop plug-in function can be regarded as a callback function, and part of the Photoshop process, Photoshop doesn't keep the global variables for when the callback function `main_entry()` is reentered. The only way provided by Photoshop to keep global variables is to use the `data` argument in `main_entry()`. We can use it to allocate a pointer to a data structure which includes the necessary information.

`Process_Image()` calls the relevant PIPT routine depending on the selection of the user, it has the following prototype:

```
IMAGE* Process_Image(IMAGE* in, GHdl globals,
                     int *param_error, int *result_error);
```

The arguments have the following meanings:

- `in` The input image to be processed.
- `globals` Stores the global information which includes the user selections, which PIPT routine is selected, the number of the workers, whether or not use timer etc.
- `param_error` The returned value. In `Process_Image()`, the input parameters for the image processing are analyzed. Only valid input parameters can be passed to PIPT routines. Otherwise, this argument would return nonzero.
- `result_error` If it is nonzero, then there is error during the image processing. It is a returned value.
- When the image is processed successfully, the output image is returned in the form of a pointer to an `IMAGE` data structure as defined by the PIPT. A NULL pointer is returned otherwise.

Because the feature routines in the PIPT have their own characteristics, we designed a specific function `Do_Features()` for them. The details of how we deal with the features is introduced in Section 7.3.2. The following is its prototype:

```
FEATURE* Do_Features(IMAGE* in, GHdl globals, int *param_error);
```

The arguments have the following meanings:

- `in` The input image to be processed.
- `globals` The same as in function `Process_Image()`.
- `param_error` The same meaning as in `Process_Image()`.
- When the image is processed successfully, the output `FEATURE` data structure as defined by the PIPT is returned, otherwise it is NULL.

The Graphical User Interface

The GUI is written using the X/Motif libraries. Some of the resources used are written in UIL (User Interface Language). According to the way of parameter input of each PIPT routine, the GUI is classified into four kinds, the main PIPT plug-in GUI shown in Figure 7.2, the input GUI, the output GUI and the GUI for specific PIPT routines.

The main PIPT plug-in GUI is a dialog box realized by the function `PIPT_UI()` which has the following prototype:

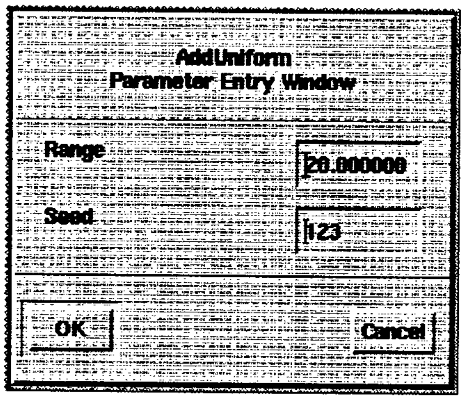
```
void PIPT_UI(int* type, int* filter, int* procs_num, int* timer);
```

The arguments have the following usage,

- `type` The type of the PIPT routine that the user selects. It is one of the five values defined in the header file `PIPT.h`: FILTERS, FEATURES, IMAGEHANDLING, ENHANCE, RANDOM.
- `filter` The PIPT routine selected by user such as IPAVERAGE, IPHISTOGRAMEQUAL, etc.
- `procs_num` The number of workers to process the image. When the PIPT is initialized, it passes the number of processors available to do work into the function and returns the number of workers to process the image.

- `timer` Whether to turn on the timer which calculates the image processing time. If it is 1, turn the timer on, if 0, off.

The second and third GUI window used in the PIPT plug-in are the PIPT plug-in *Parameter Input* and *Result Output* dialog. Because almost all the PIPT routines need to input relevant parameters a standard dialog box pops up when user chooses one of the PIPT routines from the main PIPT plug-in dialog box. In this dialog the user can choose the parameters, then click the "OK" button to process the image. The parameter input and result output of almost all the PIPT routines are very similar, so two functions are designed to realize these input and output dialogs. One is called `Input_UI()`, it creates the input dialog boxes of the PIPT routines. Figure 7.3 shows an example used by `IPAddUniform()`. There are two parameters needed by this routine, one is `Range` the other is `Seed`.

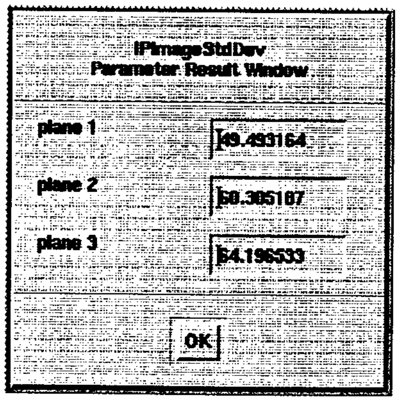

AddUniform
Parameter Entry Window

Range 20.000000

Seed 123

Figure 7.3: Input dialog box

Because the output of some PIPT routines is just a few numbers, such as in `IPImageMean()`, `IPImage-StdDev()` etc, there is another `Output_UI()` function to create an appropriate dialog. It creates the output dialog box to display the resulting data. Figure 7.4 shows an example of such dialog box.


IPImageStdDev
Parameter Result Window

plane 1 49.493164

plane 2 60.305107

plane 3 64.196533

Figure 7.4: Output dialog box

These two functions have the following prototypes:

```
int Input_UI (char *DialogClass, char *DialogTitle,
              char *DialogLabel, CallbackData *Data)
void Output_UI (char *DialogClass, char *DialogTitle,
                char *DialogLabel, CallbackData *Data)
```

The arguments used in these two functions have the following meanings:

- `DialogClass` The name of the dialog class in X/Motif programming.
- `DialogTitle` The title of the dialog box.
- `DialogLabel` The label shown in the upper area of the dialog. For example in Figure 7.3, the string 'AddUniform Parameter Entry Window' is the label.
- `Data` It has the data structure `CallbackData` which is introduced in Section 7.3.2. Depending on `Data`, the function can arrange to create the correspondent dialog for the PIPT routine . `Input_UI()` is filled with the parameters needed by the PIPT routine. In `Output_UI()` it is filled with the output result data. The `CallbackData` structure is introduced Section 7.3.2.
- `Return value` If user clicks "OK" button, it returns 1, and the image is processed. If user clicks "Cancel" button, it returns 0, and the image is not processed.

The last kind of PIPT GUI window is for some routines in features type, such as `IPCoOccurCluster()`, `IPCoOccurContrast()` etc. The function `CoOccur_Input_UI()` is designed to create the dialog box shown in Figure 7.5. It is defined as the following:

```
int CoOccur_Input_UI(char *dialogtitle, char *dialoglabel,
                     CallbackData *Data);
```

The arguments used in this function have the same meanings as `Input_UI()`.

Besides the above functions, another two functions `ImageMoments_UI()` and `ImageHistogram_UI()` are designed for `IPImageMoments()` and `IPImageHistogram()` routines respectively. They have the following prototypes:

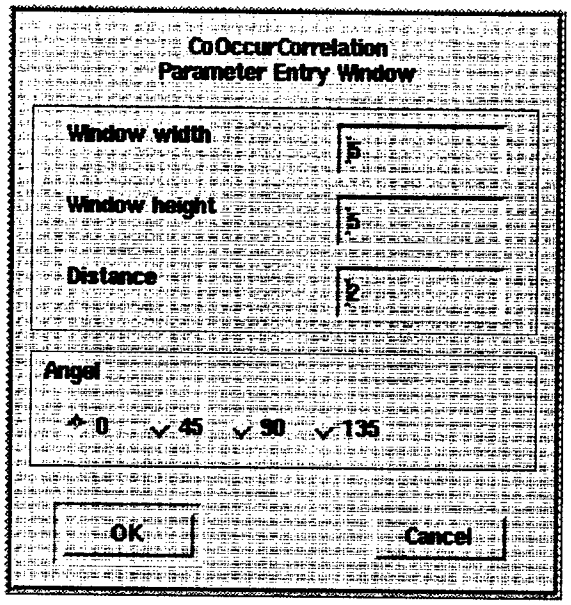


Figure 7.5: IPCoOccur dialog box

```
void ImageMoments_UI(unsigned int in_moment, unsigned int in_plane,
                     double **data);
void ImageHistogram_UI(unsigned int in_plane, unsigned long **data);
```

The arguments in the two functions are correspondent to `IPImageMoments()` and `IPImageHistogram()`.

Except `PIPT_UI()` is in the file `PIPT_Private_UI.c`, all the other functions are in the file `PIPT_Public-_UI.c`. Each file has a related UIL resource file. They are `PIPT_Private_UI.uil` and `PIPT_Public-_UI.uil` respectively.

The Interface With the PIPT

For programs that use MPI, we must use `mpirun` [24] to start the program instead of the normal method of starting the application program. Because Photoshop is not designed to work with MPI, we can not run Photoshop using `mpirun` in order to call the PIPT plug-in. To avoid using `mpirun`, we add the function `PIPT_Spawn_Init()` which does the same things as `PIPT_Init()` in the PIPT. In `PIPT_Spawn_Init()`, `MPIL_Spawn()` [19] is called to spawn the child processes `PIPT_Child` in the universal machines. [1] They set up the environment needed by the PIPT.

`PIPT_Spawn_Init()` is designed as the following:

[1]Only the LAM (Local Area Multicomputer) [6] implementation of MPI has the function `MPIL_Spawn()`, it is not a standard function. This is the reason why we use the LAM version of MPI in the PIPT plug-in.

```
void PIPT_Spawn_Init(int Userargc, char *Userargv[],
                     char *app, int *proc_num);
```

The arguments have the following usages:

- `Userargc, Userargv` They are the same as `PIPT_Init()`. For the LAM version MPI, argc and argv are not really needed by `MPI_Init()`, and the callback function `main_entry()` doesn't have argc and argv, so we create them in the PIPT plug-in artificially.
- `app` This is a character string used in calling `MPIL_Spawn()`. It decides how to spawn the child processes `PIPT_Child`. The format of this string can be seen in `mpi` man page.
- `procs_num` After `PIPT_Spawn_Init()` is called, it returns the universal size of processors.

The call to `MPIL_Spawn()` in `PIPT_Spawn_Init()` is shown in figure 7.6.

```
MPI_Comm intercom, parent, comm_world;
if (app != NULL) {
  MPIL_Spawn(MPI_COMM_WORLD, app, 0, &intercom);
  MPI_Intercomm_merge(intercom, 0, &comm_world);
}
else {
  MPIL_Comm_parent(&parent);
  MPI_Intercomm_merge(parent, 0, &comm_world);
}
```

Figure 7.6: Call to MPIL_SPAWN

If app is not NULL, a child processes is spawned according to app. This is the job of the manager. If app is NULL, the process must be a child. The parent processes is obtained and the communicator `comm_world` is created.

The child process is a very simple program, it has three lines and only calls `PIPT_Spawn_Init()` and `PIPT_Exit()`. Figure 7.7 shows the source code of `child.c`:

As stated previously, the LAM implementation of MPI is required because Photoshop cannot be used in conjunction with `mpirun`. The LAM implementation of MPI requires some setup for it to function. Before the PIPT plug-in is selected from the Photoshop `Filter` menu, the LAM parallel machine must be started using `lamboot`. After the user is finished, `wipe` must be run to shut down LAM. Please refer to the LAM [6] documentation for information about installing LAM and using `lamboot`, `wipe`, etc.

```
#include <pipt.h>

int main(int argc, char **argv)
{
  int nprocs;
  PIPT_Spawn_Init(argc, argv, NULL, &nprocs);
  PIPT_Exit();
}
```

Figure 7.7: Child Program

Feature Handling

The PIPT defines its own feature data structure. It provides a function called `IPFeatureToImage()` to convert features to images. Each feature can be converted into the same number of images as the number of planes. Each image has only one plane (grayscale image). Such as if a image is RGB which has 3 planes, its output feature will be converted into 3 grayscale images. But the Photoshop filter plug-in doesn't provide the interface to display several images at the same time, so we have to combine these converted images into one image in order to display the feature result visually. The way of combining the images into one image is shown in Figure 7.8.

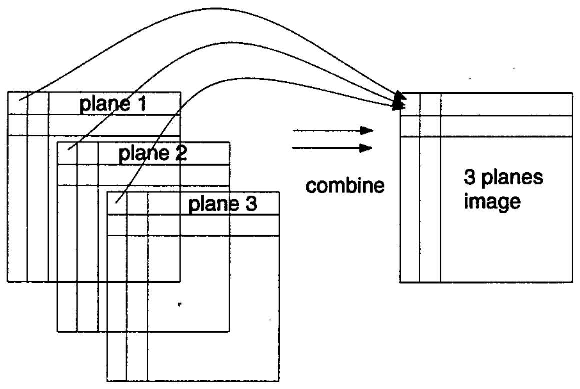


Figure 7.8: Image combination

Data Structure

Several data structures are designed for the PIPT plug-in. They are defined in the header file `PIPT.h`. For the GUI of the PIPT plug-in, a `CallbackData` data structure is defined (Figure 7.9). It is used in the functions `Input_UI()`, `Output_UI()` etc.

```
typedef struct _CallbackData {
  int itemnum;
  Item item[MAX_NUM_WIDGET];
} CallbackData;

typedef struct _Item {
  char *labelname;
  char *classname;
  int  type;

  int   value_int;
  float value_float;
  char* value_string;
} Item;
```

Figure 7.9: Callback Data Structure

The element `itemnum` is the number of parameters. The `item` is an array which has the `struct` type, it holds the information used in the GUI for each parameter. The elements in this structure have the following usage:

- `labelname` The name of the input parameter shown in the dialog box.
- `classname` The class name of the parameter resource.
- `type` Indicate which type the item is, there are three types `INT`, `FLOAT`, `STRING`. `INT` represents integer, `FLOAT` represents `double`, `STRING` represents character string.
- `value_int` When the `type` is `INT`, it is the integer value of this item.
- `value_float` When the `type` is `FLOAT`, it is the double value of this item.
- `value_string` When the `type` is `STRING`, it is the character string value of this item.

Since Photoshop interferes with the use of global variables, two data structures `TParameters` and `Globals` are created in order to keep the parameters related to the GUI and other information. The definition of these data structures is shown in Figure 7.10.

In `TParameters`, there is just one element which is `CallbackData` type. In the function of the selector `filterSelectorParameters`, the `parameters` in the `FilterRecord` is initialized and allocated a new handle to store the `CallbackData`.

Some necessary information is included in `Globals` except the `FilterRecord`, they are the followings:

```
typedef struct _TParameters {
  CallbackData Data;
} TParameters, *PParameters, **HParameters;

typedef struct Globals
{
  short result;
  short type;
  short routine;
  short workers;
  short timer;
  short pipt_init;

  FilterRecord *stuff;
} Globals, *GPtr, **GHdl;
```

Figure 7.10: Globals Data Structure

- `result` Stores the result of user selection and whether there is error in image processing.
- `type` The type of the PIPT selected routine; it is one of the five values, FILTERS, ENHANCE, IMAGEHANDLING, RANDOM, FEATURES.
- `routine` The selected routine such as IPAverage, IPCrossMedian etc ...
- `workers` The number of workers to process the image.
- `pipt_init` The flag of initializing the PIPT, it is not used in the current version of the PIPT plug-in.
- `stuff` The information related to filter plug-in, `FilterRecord` is defined by Photoshop plug-in, we have introduced it in Section 7.3.1.

When the plug-in is selected for the first time in Photoshop, we use the argument `data` passed by the callback function `main_entry()` to allocate a handle to keep the globals.

File Format

Besides all the image format files supported by Photoshop, there are two kinds of files used in the PIPT plug-in. One is the feature file used in the feature processing, the other is the data input file for `IPFeatureSepConvolution`. These two files are all text format files. The format of the feature file is :

```
File Header:            ``IPToolkitFeatureFileIDVersion1'' (string type)
Feature parameters:     dimension, height, width (long integer type)
Feature data:           data, data, data, . . . (float type)
```

The format of the input data file for `IPFeatureSepConvolution` is comma separated floating point numbers.

Using `diff` provided by the UNIX system, we compare the output image files generated by the PIPT plug-in and the PIPT respectively for the same PIPT routines and the same input parameters. `diff` reports that the two files are the same. This confirms that the PIPT plug-in produces correct output. From the view of parallelism, the PIPT plug-in achieves a nearly linear speedup which is similar to the PIPT.

Because the Photoshop plug-in doesn't permit us to change the size and the number of planes of the image, there are six PIPT routines which are not included in the plug-in. Five of them are of the Imagehandling type, one of them is of the Features type. Almost all of them try to change the size or the number of the planes of the image. The following shows the reasons why they are not part of the PIPT plug-in:

- `IPCopyImage()`: Photoshop can do it well.
- `IPImageFlipHorizontal()`: the size of the image is changed.
- `IPImageScale()`: the size of the image is changed.
- `IPImageToGray()`: the number of planes is changed.
- `IPImageToImage()`: the number of planes can be changed
- `IPCannyEdge()`: the number of the feature planes is always 2.

`IPCannyEdge` is Features type, all the others are Imagehandling type.

Hopefully, these routines can be added in the next version of the PIPT plug-in.

7.4 Conclusions

The PIPT plug-in inserts the PIPT into Adobe Photoshop successfully and it realizes parallel image processing in a widely used image processing software application. There are a few issues that went unaddressed in this initial version of the PIPT plug-in. These will be dealt with in two improvements for the next version of the PIPT plug-in. These future improvements are beyond this effort.

7.4.1 Change of Image Data Format

Right now, the PIPT and Photoshop use different data formats to store the image data. When image data is passed between the PIPT and Photoshop, we need to change the data format. This is done sequentially and can represent a significant overhead. Therefore we will change the data format in the next version of the PIPT plug-in and eliminate the steps used to translate the format.

7.4.2 Improvement in PIPT Initialization

With the PIPT plug-in, the PIPT has to be initialized each time the PIPT plug-in is selected. This is because global variables can not be used from a Photoshop plug-in. It will take some time to set up the environment needed by the PIPT. So in the next version the PIPT plug-in, a way will be figured out to initialize PIPT only once when the user invokes the PIPT plug-in the first time.

7.4.3 Probable Bug in Photoshop

When the input image is a grayscale image, and the width of the selected area is not the same as the width of the image, the image processing results are incorrect. The reason is that Photoshop does not pass the image data to the plug-in program properly. Instead of passing just the region we are interested in, for each row in the region, the entire row is passed in, not just the columns that are part of the region. Since more data than is desired is passed, and the computation uses the extra data, and operates on this new super region, the final result is incorrct by a small factor, and more than the desired region was processed on. Moreover, not just the region, but again all the rows contained in the region, are updated with the result, which overwrites portions of the image which the user had not intended to operate on.

7.5 Running the PIPT Plug-in

Before running Photoshop, the PIPT plug-in must be installed, otherwise you will not find the submenu `PIPT` under the menu `Filter` of Photoshop. If the PIPT plug-in has already been installed, there is a submenu `PIPT` under the `Filter` menu in Photoshop.

Before the PIPT plug-in is invoked from the menu `Filter` in Photoshop, LAM/MPI *must* be started. Otherwise the behavior is undefined. To start LAM, execute the following steps:

1. The directory where LAM binaries are installed *must* be in your `.cshrc` path.

2. A text file listing machines to be used must be created. The "host" machine (i.e., the machine that the jobs will be run from) must be listed first in the file. For best performance, use machines on one subnet.

3. LAM/MPI must be started on the target machines. From the command line, type:

   ```
   unix% lamboot -v hostfile
   ```

   where hostfile is the filename of the file containing the hostnames of the machines to be used.

4. After LAM has been started (`lamboot` spawns a daemon on all the machines in the hostfile), you can verify that they are working properly with the command:

   ```
   unix% tping N -c3
   ```

   If LAM is operating correctly, you will see the results of a "ping" type of command that visits each machine in the LAM boot.

After LAM is started, the PIPT plug-in can be selected from the submenu `PIPT` under the `Filter` menu in Photoshop. A dialog box will appear, then you can choose the PIPT routine to process the image.

If you are finished using the PIPT plug-in, LAM may be shut down. This is accomplished with the `wipe` command and the same hostfile that was previously used with the `lamboot` command to start LAM:

```
unix% wipe -v hostfile
```

Chapter 8

Summary and Conclusions

There were a number of important results obtained over the course of this work. In this chapter, we summarize the concrete accomplishments of this effort, draw some conclusions, and suggest some directions for future work.

8.1 Summary

8.1.1 Advanced Data Handling

Re-evaluating the PIPT code to take advantage of advanced data handling techniques such as keeping microprocessor pipelines full, reducing cache misses, and multithreading proved to be a wide-reaching task. The PIPT internal code was re-structured to take advantage of advanced data handling techniques and to force certain types of compiler optimizations, especially within loops over arrays of data. Nearly every image and feature processing routine was modified; all kernels were updated.

The existing structure of the PIPT greatly facilitated the inclusion of various optimizations. Many of the image processing routines in PIPT make use of a small number of fundamental computational kernels. Concentrated optimization of these kernels provided increased performance throughout PIPT. In addition, the fundamental kernels typically operate with a helper processing function (e.g., `IPAverage()` uses the `ProcessWindow()` kernel in conjunction with a particular `ComputePoint()` helper function). Additional optimizations, on a per-function basis, were obtained by optimizing these helper functions. Finally, the mechanisms for manipulating data between the kernels and the helper functions (e.g., data copying) were also optimized.

Compiler and underlying hardware issues, as with any software package intended for multiple platforms, continued to cause portability problems. The PIPT was not portable to DEC OSF versions of Unix prior to version 4.0. However, through selective compilation, the PIPT remains portable to a variety of hardware and software platforms.

Further performance optimizations could likely be obtained in two ways: by moving to a more optimizable programming paradigm and by making use of microprocessor-specific instructions (such as MMX or VIS). The mechanisms used by the PIPT to provide genericity in the PIPT implementation also tend to interfere with certain optimizations. For example, many of the kernels repeatedly call functions through pointers within an inner computational loop. This overhead could be removed by compile time polymorphism, such as that available in the C++ template system. Some of these issues related to generic programming and compile-time polymorphism are discussed in more detail below.

8.1.2 Load Balancing

The First-Finish, First-Serve (FFFS) algorithm proved to be an effective load balancing algorithm, particularly in active heterogeneous workstation environments. The ability to adjust the FFFS parameters at run-time provides users with an additional tool to maximize performance in a variety of different environments. But under some circumstances (particularly in active heterogeneous workstation environments), FFFS can get stalled by slow or loaded processors. The Redundant FFFS (RFFFS) algorithm, based upon the original FFFS algorithm, solves this problem by allowing multiple processors to work on the same slice.

Unfortunately, since currently the LAM implementation of MPI is not thread safe, allowing redundant processors to abort their current slice when the slice has already been returned by another process to the manager was not possible. Future implementations of LAM may include some degree of thread safety, which would allow this optimization to the RFFFS implementation. It should be noted that while other implementations of MPI are thread safe, these implementations are only provided by vendors for their specific architectures. LAM was chosen for this project because it works over clusters of heterogeneous workstations; it provides the maximum flexibility for execution environments.

Further modifications to the load balancing schemes, to include sending non-uniformly sized slices and/or multiple slices to worker nodes, were investigated but not implemented because such enhancements require a different encapsulation model than is presently included in the PIPT. These enhancements would have been particularly useful for multiprocessor workers.

Analysis of network patterns shows that faster networks (such as 100Mbps ethernet and 155Mbps ATM) significantly decrease overhead time, and therefore sped up the overall computation.

8.1.3 Parallel HRVS

Implementing a parallelized version of the HRVS system was a specific requirement of this effort. Incorporating HRVS into the PIPT framework illuminated a number of characteristics of the PIPT. First, since HRVS is a computationally intensive algorithm, the speedups obtained with the parallelized version demonstrated the effectiveness of parallel computing and, to some degree, the flexibility and power of the PIPT toolkit. On the other hand, there were some aspects of the HRVS algorithm that did not completely fit the PIPT image processing model. This is not a particular shortcoming of the PIPT framework. The PIPT was designed with image processing in mind, not video processing. Since the HRVS algorithm works with video data, it is no surprise that it was not completely straightforward to implement HRVS using PIPT. Future extensions to the PIPT should include mechanisms for handling video data in general, and for the HRVS in particular.

8.1.4 Parallel Visualization

Because of the low-level nature of visualization, it is not feasible to implement a portable parallel visualization library from the bottom up. That is, at the lowest level, the library will, by necessity, have to work directly with the particular video hardware and microprocessor architecture on the execution target. Thus, the parallel visualization library will need to have at least two layers: the low-level vendor- and environment-specific layer, and the application interface layer (consistent across all platforms).

In this effort, we designed and implemented the general interface layer for the parallel visualization library and provided a single instance of the low-level specific layer. In this case, the specific target was Sun SPARC Solaris machines. Fortunately, the low-level layer was itself a high-performance vendor-supplied visualization library, making it portable across most of the recent Sun Microsystems platforms.

Porting to other platforms can be accomplished by implementing low-level layers for those particular platforms. Since most vendors have an interest in providing high-performance visualization, one can expect that most target environments will have at least some rudiments of vendor-tuned routines to hook into.

8.1.5 Interface to Adobe Photoshop

Having a visualization interface to the PIPT would seem to be an important and powerful extension to the library. Initially, interfacing to Adobe Photoshop seemed promising because Photoshop is a powerful and robust visualization environment and one that is extensible via its plug-in technology. There are several different levels at which one can plug-in to Photoshop. Unfortunately, the level at which it would make the most sense to plug in PIPT is only available through special licensing agreements with Adobe — and Adobe is no longer adding any new developers to this particular program.

Thus we were left with having to plug in the PIPT to Photoshop as a filter. While this approach did work, filter plug-ins are loaded and unloaded each time they are invoked, so that the PIPT has to be completely initialized each time a PIPT operation is selected. This puts certain constraints on the run-time environment of the PIPT plug-in (not to mention the obvious lack of elegance in such an approach).

A more fruitful approach for a visualization front-end for the PIPT may be to interface to a public domain visualization environment such as the GNU Image Manipulation Program (GIMP). Besides having various types of plug-in technology, the GIMP is distributed as source code, so that, if necessary, modifications to the visualization program itself could be made to accommodate the PIPT interface.

8.2 Conclusions

There were several important contributions that resulted from this work and which in some sense transcend the particular tasks that were undertaken. First, the general approach that was taken to implementation of the PIPT illustrates several important principles for the design and implementation of general purpose parallel libraries. In this regard, the PIPT design can serve as a "design pattern" for an extensible parallel library. Second, the design pattern of the current implementation of the PIPT contains some notable attempts at programming image processing tasks in a generic fashion.

Below, we discuss the design pattern evolved by PIPT as well as the issues involved in generic programming for image processing.

8.2.1 A Design Pattern for Task Farm Parallelism

The design pattern evolved by PIPT can be summarized as being a model for "task-farm" parallel computation. That is, computations are doled out to a number of worker processes. The computations are assigned in the form of a function to execute, along with input parameters to the function. The worker executes the specified function and sends back the result.

The value added in this work is in the framework that allows this type of parallelization to be easily accomplished for large classes of new functions. PIPT provides a simple registration-callback mechanism by which one can define new functions to the system. Once the function to be parallelized is specified to the parallel execution system, the underlying transport layer takes care of marshaling data to and from the worker processes and performing the actual parallel computations. The user is thus shielded from many of the typical concerns about parallel programming. A diagram of the PIPT architecture is shown in Figure 2.7.

8.2.2 Generic Programming as an Approach for High Performance

The computational kernel framework within PIPT is a reasonable design for genericity. That is, PIPT provides certain fundamental computational kernels that accomplish generic tasks, e.g., `ProcessWindow()` applies a specified window operation repeatedly across the input image. Different high-level image processing functionality is realized by calling `ProcessWindow()` with different window operations.

One drawback to this approach is that there is a certain overhead involved in invoking the window operation (specified by a function pointer) within the inner loop of `ProcessWindow()`. However, the use of function pointers is basically the only way to obtain this type of programming flexibility with the C language.

In C++, however, there are a number of alternatives. One approach is to use inheritance, but that basically just hides the function pointer, the polymorphism is still accomplished at run-time. A more effective approach is to take advantage of compile-time polymorphism that is provided by the C++ template system, which allows the function to be inlined, thus removing the overhead.

In this case, the prototype for `ProcessWindow()` might look like the following:

```
template <class Image, class Region, class Operator>
void
ProcessWindow(const Image& ImageIn, Image& ImageOut,
              Operator computePoint);
```

The body of `ProcessWindow()` would be similar to that of the Standard Template Library (STL) [21] `transform()` algorithm and would consist of an iteration over the input image pixels to produce the output image pixels. The `computePoint()` argument could be a function pointer or (more generally) a function object. The use of iterators would further generalize this framework and would provide a ready mechanism for handling different image formats with no changes to the processing algorithms.

8.3 Future Work

Continuing research in image processing will entail a change in several of the internal PIPT models to afford optimizations that are not possible with the current implementation. These changes naturally fit within an object-oriented design, and would most easily be implemented by converting the PIPT to C++. C++ has significantly more expressive power, offers better data encapsulation than C, and allows for compile-time binding of image processing functions (vs. run-time binding with function pointers) through inlining and templates. Compile-time binding of functions will allow for much greater compiler optimization of image processing loops. The development of a generic framework for image processing is an exciting new avenue and should be more fully explored.

Hyperspectral images may also be incorporated into the PIPT; the ability for optimized hyperimages with 100's or 1000's of data planes rather than single- or triple-planed images.

Parallel input/output opportunities may be investigated with the recent implementation of the MPI-IO library from the Argonne National Laboratory. Thread safety issues may also continue to be explored; in particular cases, it is possible to utilize non-thread-safe libraries in a multi-threaded environment.

Finally, the possibility for asynchronous one-sided operations may be researched as implementations of the one-sided MPI functions become available.

Bibliography

[1] Adobe. Adobe photoshop - software development kit documentation - macintosh version, 1993.

[2] Adobe. Adobe photoshop for unix platforms plug-in developement notes, 1995.

[3] Adam Beguilin et al. A users' guide to PVM parallel virtual machine. ORNL/TM 11826, Oak Ridge National Laboratories, Oak Ridge, TN, 1992.

[4] Jeff Bilmes, Krste Asanovic, Jim Demmel, and C-W. Chin. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, Vienna, Austria, July 1997.

[5] R. Butler and E. Lusk. User's guide to the p4 programming system. TM-ANL 92/17, Argonne National Laboratory, Argonne, IL, 1992.

[6] Raja Daoud, Greg Burns, and Nick Nevin. *Local Area Multicomputer (LAM) User's Manual.* Ohio Supercomputing Center / University of Notre Dame, Notre Dame, IN, October 1998.

[7] Nathan E. Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. An initial implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[8] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[9] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.

[10] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: A portable instrumented communication library. ORNL/TM 11616, Oak Ridge National Laboratories, Oak Ridge, TN, October 1990.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[12] John L. Hennessy and David A. Patterson, editors. *Computer Architecture - A Quatitative Approach.* Morgan Kaufmann Publishers, Inc, 1996.

[13] IEEE. *An American National Standard: IEEE Standard for Binary Floating-Point Arithmetic*, volume 754-1985. IEEE, 1985.

[14] Kuck & Associates, Inc., Champaign, IL. *KAI C++ Documentation Set*, 1996.

[15] Brian C. McCandless, John J. Tran, Jeffrey M. Squyres, Andrew Lumsdaine, and Robert L. Stevenson. *Software Documentation Parallel and Distributed Algorithms for High-Speed Image Processing.* Laboratory for Scientific Computing, University of Notre Dame, Notre Dame, IN, 1995.

[16] R. R. Schultz and R. L. Stevenson. A bayesian approach to image expansion for improved definition. *IEEE Transactions on Image Processing*, 3(3):233–242, 1994.

[17] R. R. Schultz and R. L. Stevenson. Extraction of high-resolution frames from video sequences. *IEEE Transactions on Image Processing*, 1995.

[18] Anthony Skjellum. *Concurrent dynamic simulation: Multicomputers algorithms research applied to ordinary differential-algebraic process systems in chemical engineering.* PhD thesis, California Institute of Technology, May 1990.

[19] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W.Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, Cambridage, Massachussetts, 1996.

[20] Jeffrey M. Squyres. MPI: Extensions and applications. Master's thesis, University of Notre Dame, Notre Dame, IN, 1996.

[21] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[22] SunSoft, Inc. *XIL Programmer's Guide*, 1994.

[23] SunSoft, Inc. *XIL Reference Manual*, 1994.

[24] Ewing Lusk William Gropp and Anthony Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridage, Massachussetts, 1994.

Appendix A

Test Results

A.1 Test Plan Definition

This document contains the specifications for the acceptance test plan for the Parallel Image Processing Toolkit, as well as for two related packages, the High-Resolution Video Stills (HRVS) library and the Parallel Visualization (PVIZ) library.

Unless otherwise specified, the terms "the delivered software" should be understood to include the PIPT 2.1 and the contributed HRVS and PVIZ libraries.

The following particular aspects of the delivered software have been tested:

Conformance of delivered source code: The source code must compile with no warnings and run with no memory leaks.

Functionality: All PIPT 2.1, HRVS, and PVIZ image processing routines must function correctly, as specified by this test plan.

Sequential and parallel performance: Sequential and parallel performance of the PIPT 2.1 should be improved in comparison to the PIPT 1.0.3. Parallel performance of HRVS and PVIZ should show reasonable scalability.

Each of these aspects of the test plan are described in more detail below.

A.2 Source Code Conformance

There are two aspects to verifying the conformance of the delivered software. First, the software must compile on all specified architectures and operating systems with no compiler warning messages, with any exceptions fully explained and justified. Second, the code must execute with no memory leaks, again, with any exceptions fully explained and justified.

A.2.1 Compilation Environments

Software has been tested on the system configurations listed in Table A.1. A "$\surd$" entry indicates that the PIPT ported successfully to that architecture, while a "$\otimes$" entry indicates that the PIPT was not successfully ported to the specified architecture.

Architecture	**Operating System**	**Compiler**	**Result**
Sun SPARC / UltraSPARC	Solaris v2.4	gcc v2.7.2.3	$\surd$
		Solaris cc v4.2	$\surd$
	Solaris v2.5.1	gcc v2.7.2.3	$\surd$
		Solaris cc v4.2	$\surd$
	Solaris v2.6	gcc v2.7.2.3	$\surd$
		Solaris cc v4.2	$\surd$
Dec Alpha / Alpha Server	OSF v2.0	gcc v2.7.2.3	$\otimes$
	OSF v3.2	gcc v2.7.2.3	$\otimes$
	OSF v4.0	OSF cc v5.2-036	$\surd$

Table A.1: System configurations for software testing

The software must compile on the above supported configurations with no compiler warning messages; any exceptions will be fully justified and explained.

- The PIPT software failed to port to OSF v2.0 because the operating system does not support standard IEEE floating point arithmetic. Therefore, output from the test suite never agreed with output from other architectures. By our defintion, this constituted incorrect answers. There was no way to fix this problem within that release of the operating system. The PIPT has been declared incompatible with OSF v2.0.
- The PIPT software failed the test suite on OSF v3.2 for the same reasons as listed above. There was no way to fix this problem within that release of the operating system. The PIPT has been declared incompatible with OSF v3.2.

A.2.2 Memory Leaks

The software must also run Solaris `bcheck` with no memory leaks reported. Since the Solaris `bcheck` program can only operate for non-distributed programs, the PIPT was only tested for memory leaks while running on one node only. Since the single node code is identical to the distributed, single node conformance should be sufficient to guarantee the absence of memory leaks in the distributed memory case.

Any exceptions (i.e., any leaks reported by `bcheck`) must be fully justified and explained. An example of such an exception would be a memory leak that occurs in a third-party library (such as the `tiff` image library).

A.3 Functionality Testing

To test functionality, we exercise each PIPT image processing function with a variety of parameters and verify that correct output is produced. These results were generated with the following test environment:

- Sun UltraSPARC 140e workstations,
- Solaris 2.5.1 operating system,
- Workshop 4.2 C compiler,
- 100baseT Ethernet connectivity, and
- LAM 6.1 version MPI.

A.3.1 PIPT 2.1

The IP Toolkit (IPT) was used to generate reference output for the enhance, feature, filter, and handling types of routines. One class of routines, random routines, cannot be tested against output from the IPT because the PIPT uses a different random number generator than the IPT (making exact comparison meaningless). Random routines were tested against the output of PIPT 1.0.3.

A `contrib/Test_suite` subdirectory has been added to the PIPT code tree for the purposes of testing functionality. This test suite generated output using the PIPT 2.1, which was compared against the correct output previously generated (from the IPT). To check the full operability of the software, each test was repeated 10 times with two different input images (one color, one greyscale) to ensure persistence. The two test input images, `cars.tif` and `eggs.tif`, are shown as Figures A.1 and A.2 in Appendix A.5, respectively. `cars.tif` is an 896×636 black and white image; `eggs.tif` is a 258×220 color image. Both input images were used as input for every tested routine. These cases are shown in Tables A.2 through A.9.

Routine	**Parameter Values**	**# Nodes**	**Result**
IPCenterMean	whght = 5, wwdth = 5, fweight = 5	1	√
		5	√
		8	√
IPContrastAdjust	fAdjustParam = 0.5	1	√
		5	√
		8	√
IPGammaAdjust	dGamma = 3.0	1	√
		5	√
		8	√
IPHistogramEqual	none	1	√
		5	√
		8	√
IPIntensityMap	pixelmap[1]	1	√
		5	√
		8	√
IPInvert	none	1	√
		5	√
		8	√
IPStretchRange	none	1	√
		5	√
		8	√
IPUnSharpMask	none	1	√
		5	√
		8	√

Table A.2: PIPT enhancement routines

[1]`pixelmap` is a monotonically decreasing 256 element array; the first element's value is 255.

Routine	Parameter Values	# Nodes	Result
IPATrimmedCrMean	win_height = 5, win_width = 5, alpha = 1.0	1	√
		5	√
		8	√
IPATrimmedSqMean	win_height = 5, win_width = 5, alpha = 1.0	1	√
		5	√
		8	√
IPAverage	win_height = 5, win_width = 5	1	√
		5	√
		8	√
IPConvolution	infeature = small.feat[2]	1	√
		5	√
		8	√
IPCrossMedian	win_height = 5, win_width = 5	1	√
		5	√
		8	√
IPGaussianSmooth	stddev = 1.0	1	√
		5	√
		8	√
IPSquareMedian	win_height = 5, win_width = 5	1	√
		5	√
		8	√

Table A.3: PIPT filter routines

[2] `small.feat` is generated by calling IPAboveThreshold on a 13 by 13 tiff cropped from the grayscale `cars.tif`.

Routine	Parameter Values	# Nodes	Result
IPAboveThreshold	whght = 5, wwdth = 5, threshold = 150	1	√
		5	√
		8	√
IPBelowThreshold	whght = 5, wwdth = 5, threshold = 150	1	√
		5	√
		8	√
IPCannyEdge	aspect = 1.0, scale = 1.0	1	√
		5	√
		8	√
IPCoOccurCluster	whght = 5, wwdth = 5, dist = 4, angle = 2	1	√
		5	√
		8	√
IPCoOccurContrast	whght = 5, wwdth = 5, dist = 2, angle = 2	1	√
		5	√
		8	√
IPCoOccurCorrelation	whght = 5, wwdth = 5, dist = 0, angle = 3	1	√
		5	√
		8	√
IPCoOccurEnergy	whght = 5, wwdth = 5, dist = 0, angle = 3	1	√
		5	√
		8	√
IPCoOccurEntropy	whght = 5, wwdth = 5, dist = 0, angle = 3	1	√
		5	√
		8	√
IPCoOccurHomogeneity	whght = 5, wwdth = 5, dist = 0, angle = 3	1	√
		5	√
		8	√
IPCoOccurInverseMoment	whght = 5, wwdth = 5, dist = 0, angle = 3	1	√
		5	√
		8	√
IPCoOccurMax	whght = 5, wwdth = 5, dist = 0, angle = 3	1	√
		5	√
		8	√

Table A.4: PIPT feature routines, part 1

Routine	Parameter Values	# Nodes	Result
IPFeatureConvolution	infeature = small.feat[3]	1	√
		5	√
		8	√
IPFeatureSepConvolution	fltrordr = 5, pkKernelParam.file[4]	1	√
		5	√
		8	√
IPFreiEdge	none	1	√
		5	√
		8	√
IPKirschEdge	none	1	√
		5	√
		8	√
IPMarrHildrethEdge	width = 0.5	1	√
		5	√
		8	√
IPPrewittEdge	none	1	√
		5	√
		8	√
IPRobertsEdge	none	1	√
		5	√
		8	√
IPRobinsonEdge	none	1	√
		5	√
		8	√
IPSobelEdge	none	1	√
		5	√
		8	√

Table A.5: PIPT feature routines, part 2

[3] A five element array = 0.0625, 0.0625, 0.125, 0.25, 0.5.

[4] `small.feat` is generated by calling IPAboveThreshold on a 13 by 13 tiff cropped from the grayscale `cars.tif`.

Routine	**Parameter Values**	**# Nodes**	**Result**
IPWindowMean	whght = 5, wwdth = 5	1	√
		5	√
		8	√
IPWindowMoments	whght = 5, wwdth = 5, nummoment = 3	1	√
		5	√
		8	√
IPWindowStdDev	whght = 5, wwdth = 5	1	√
		5	√
		8	√

Table A.6: PIPT feature routines, part 3

Routine	Parameter Values	# Nodes	Result
IP3GrayToColor	greenTiff, blueTiff[5]	1	√
		5	√
		8	√
IPAddConstant	constant = 3	1	√
		5	√
		8	√
IPAddImage	pimage2[6], iClip = 2	1	√
		5	√
		8	√
IPCopyImage	none	1	√
		5	√
		8	√
IPCropImage	tlrow = 0, tlcol = 0, height = 50, width = 50	1	√
		5	√
		8	√
IPImageFlipHorizontal	none	1	√
		5	√
		8	√
IPImageFlipVertical	none	1	√
		5	√
		8	√
IPImageRotate90	times = 1	1	√
		5	√
		8	√

Table A.7: PIPT handling routines, part 1

[5]Both inputs are the same as the source input image.
[6]Same as the input.

Routine	**Parameter Values**	**# Nodes**	**Result**
IPImageScale	fscale = 2.305	1	√
		5	√
		8	√
IPImageToColor	ulRedPlane = 0, ulGreenPlane = 0, ulBluePlane = 0	1	√
		5	√
		8	√
IPImageToGray	ulPlane = 0	1	√
		5	√
		8	√
IPImageToImage	planemap.nums = (0, 2, 1)	1	√
		5	√
		8	√
IPMultConstant	constant = 3	1	√
		5	√
		8	√
IPMultImage	image2[7], iClip = 2	1	√
		5	√
		8	√
IPSubImage	image2[8], iClip = 2	1	√
		5	√
		8	√

Table A.8: PIPT handling routines, part 2

[7] Same as the input.
[8] Same as the input.

Routine	Parameter Values	# Nodes	Result
IPAddBitError	fpercent = .05, iseed = 1234	1	√
		5	√
		8	√
IPAddGaussian	fmean = 5, fstddev = 5, iseed = 1234	1	√
		5	√
		8	√
IPAddImpulsive	fheight = 100, fpercent = .1, iseed = 1234	1	√
		5	√
		8	√
IPAddUniform	frange = 20, iseed = 1234	1	√
		5	√
		8	√
IPImageHistogram	none	1	√
		5	√
		8	√
IPImageMax	none	1	√
		5	√
		8	√
IPImageMean	none	1	√
		5	√
		8	√
IPImageMin	none	1	√
		5	√
		8	√
IPImageMoments	none	1	√
		5	√
		8	√
IPImageStdDev	ulnum_moments = 5	1	√
		5	√
		8	√

Table A.9: PIPT random routines

A.3.2 Parallel HRVS

A `contrib/Test_suite` subdirectory was added to the PHRVS code tree, for the purposes of testing functionality. A test program generated output using PHRVS 1.0.1. The output expanded image was compared to the ideal original image and displayed the difference by both Mean Absolute Value (MAD) and Signal to Noise Ratio (SNR). Three frames of the Notre Dame Administration Building video sequence were used as the input; the middle frame was expanded using both HRVS and PHRVS library routines. Signal to noise ratio (SNR) figures (relative to the ideal original image) for the two output images was then computed.

Since HRVS uses global minimizations, the output image of HRVS and PHRVS may not match exactly. However, the viewer usually cannot distinguish a SNR difference less than 0.1. Thus, if the SNR difference of two output images is less than 0.1, we say that these two images are similar enough, and that the result of PHRVS matches that of HRVS.

A.3.3 Parallel Visualization

The parallel visualization toolkit (PVIZ) is an interface between the PIPT and Sun's Xil foundation image processing library. A `contrib/Test_suite` subdirectory was added to the PVIZ code tree to test the interface layer between the PIPT and the Xil library. The output of each PVIZ function was compared against the output of the corresponding Xil function to test for correctness.

In addition, a demonstration subdirectory `contrib/Demo` was added to the PVIZ code tree to show the ability of the PVIZ toolkit to manipulate and display images on the screen.

A.4 Performance Testing

There are a number of aspects of the PIPT's performance that were tested. In particular:

Performance Comparison of PIPT 2.1 and PIPT 1.0.3: These tests were performed to demonstrate the speedup of the PIPT 2.1 over the previous version of the PIPT due to improved data handling techniques.

SMP Tests for PIPT 2.1: These tests demonstrated the performance gains obtained by PIPT operating in a shared memory environment.

Scalability Tests for PIPT 2.1: These tests demonstrated the speedup obtained by PIPT operating in various parallel environments.

Load Balancing Tests for PIPT 2.1: These tests demonstrated the robustness of the PIPT in non-dedicated parallel environments.

Routine Type	**Routines**	**Parameter Values**	**Result**
Enhance	IPCenterMean	none	√
	IPUnsharpMask	whght = 9, wwdth = 9, fweight = 9	√
Feature	IPCannyEdge	aspect = 1.0, scale = 1.1	√
	IPWindowMoments	whght = 9, wwdth = 9, nummoment = 7	√
Filter	IPAverage	win_height = 11, win_width =11	√
	IPSquareMedian	win_height = 9, win_width = 9	√

Table A.10: Subset of PIPT routines used for performance tests.

Parallel HRVS and Parallel Visualization: These tests demonstrate the parallel performance of the Parallel HRVS and Parallel Visualization components.

All timings were generated on the machines and networks noted in the Tables listed below; the Ultra-SPARC 140e and 170e machines were running Solaris 2.5.1, while the Ultra-Enterprise 3000 machine runs Solaris 2.6. The PIPT was compiled with Solaris Workshop 4.2 C compiler in all cases. The input image used was a 2910 × 3080 black and white, `big.tif`, shown as Figure A.3 in Appendix A.5.

Time was measured as the elapsed wall clock time between `start` and `stop` invocations of a timing function. Speedup is defined as

$$speedup = \frac{Basetime}{Newtime}$$

The specific definition of $Basetime$ and $Newtime$ is defined in each section below.

We tested the PIPT for both sequential and parallel performance increases. The performance tests were made using subset of routines defined in Table A.10.

Note: Most of the tests below involve parallel computations on multiple processors. Unless otherwise indicated, the indicated number of parallel processors refers to the number of worker nodes. The manager process is on a processor independent of the workers and is not counted as a worker node. The tests in Section A.4.5 show the results obtained when the manager is run on the same node as one of the workers.

A.4.1 Performance Comparison of PIPT 2.1 and PIPT 1.0.3

To compare the performance of PIPT 2.1 and PIPT 1.0.3, *Basetime* is generated with PIPT 1.0.3 and *Newtime* is generated with the PIPT version 2.1. The times are gathered for the system configurations shown in Tables A.11 through A.16.

Configuration	Device	Basetime	Newtime	Speedup
1 Sun UltraSPARC 140e	N/A	10.145	1.622	6.255
2 Sun UltraSPARC 140e	10bT	8.791	1.631	5.390
4 Sun UltraSPARC 140e	10bT	4.595	1.633	2.814
8 Sun UltraSPARC 140e	10bT	2.298	1.800	1.277
2 Sun UltraSPARC 140e	100bT	7.796	1.624	4.800
4 Sun UltraSPARC 140e	100bT	4.043	1.625	2.488
8 Sun UltraSPARC 140e	100bT	2.116	1.792	1.181

Table A.11: Test configuration for comparison of PIPT 2.1 with PIPT 1.0.3 for IPCenterMean, using parameter values shown in Table A.10. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

Configuration	Device	Basetime	Newtime	Speedup
1 Sun UltraSPARC 140e	N/A	104.789	34.649	3.024
2 Sun UltraSPARC 140e	10bT	82.411	27.699	2.975
4 Sun UltraSPARC 140e	10bT	46.976	19.304	2.433
8 Sun UltraSPARC 140e	10bT	32.079	11.695	2.743
2 Sun UltraSPARC 140e	100bT	60.349	21.527	2.803
4 Sun UltraSPARC 140e	100bT	34.218	12.244	2.795
8 Sun UltraSPARC 140e	100bT	27.604	6.955	3.969

Table A.12: Test configuration for comparison of PIPT 2.1 with PIPT 1.0.3 for IPUnsharpMask, using parameter values shown in Table A.10. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

Configuration	Device	Basetime	Newtime	Speedup
1 Sun UltraSPARC 140e	N/A	18362.726	5479.237	3.351
2 Sun UltraSPARC 140e	10bT	11328.066	3464.327	3.269
4 Sun UltraSPARC 140e	10bT	6009.954	2289.812	2.625
8 Sun UltraSPARC 140e	10bT	3489.789	1616.345	2.159
2 Sun UltraSPARC 140e	100bT	10517.114	3008.579	3.496
4 Sun UltraSPARC 140e	100bT	5896.387	1971.967	2.990
8 Sun UltraSPARC 140e	100bT	3182.812	1399.042	2.275

Table A.13: Test configuration for comparison of PIPT 2.1 with PIPT 1.0.3 for IPCannyEdge, using parameter values shown in Table A.10. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

Configuration	Device	Basetime	Newtime	Speedup
1 Sun UltraSPARC 140e	N/A	14641.126	10477.192	1.397
2 Sun UltraSPARC 140e	10bT	11076.523	5910.164	1.874
4 Sun UltraSPARC 140e	10bT	7538.262	3185.516	2.366
8 Sun UltraSPARC 140e	10bT	4654.274	1802.212	2.583
2 Sun UltraSPARC 140e	100bT	10385.549	5480.389	1.895
4 Sun UltraSPARC 140e	100bT	6387.522	2964.529	2.155
8 Sun UltraSPARC 140e	100bT	3546.472	1708.583	2.076

Table A.14: Test configuration for comparison of PIPT 2.1 with PIPT 1.0.3 for IPWindowMoments, using parameter values shown in Table A.10. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

Configuration	Device	Basetime	Newtime	Speedup
1 Sun UltraSPARC 140e	N/A	161.816	38.188	4.237
2 Sun UltraSPARC 140e	10bT	92.636	31.116	2.977
4 Sun UltraSPARC 140e	10bT	49.435	19.808	2.496
8 Sun UltraSPARC 140e	10bT	27.855	13.321	2.091
2 Sun UltraSPARC 140e	100bT	81.814	23.441	3.490
4 Sun UltraSPARC 140e	100bT	41.292	13.706	3.013
8 Sun UltraSPARC 140e	100bT	21.315	8.182	2.605

Table A.15: Test configuration for comparison of PIPT 2.1 with PIPT 1.0.3 for IPAverage, using parameter values shown in Table A.10. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

Configuration	**Device**	**Basetime**	**Newtime**	**Speedup**
1 Sun UltraSPARC 140e	N/A	538.702	413.755	1.302
2 Sun UltraSPARC 140e	10bT	281.999	248.995	1.133
4 Sun UltraSPARC 140e	10bT	143.933	125.304	1.149
8 Sun UltraSPARC 140e	10bT	74.753	64.892	1.152
2 Sun UltraSPARC 140e	100bT	269.073	232.846	1.156
4 Sun UltraSPARC 140e	100bT	135.216	118.080	1.145
8 Sun UltraSPARC 140e	100bT	67.876	57.268	1.185

Table A.16: Test configuration for comparison of PIPT 2.1 with PIPT 1.0.3 for IPSquareMedian, using parameter values shown in Table A.10. Here, *Basetime* is the time obtained by PIPT 1.0.3 and *Newtime* is the time obtained by PIPT 2.1.

A.4.2 Scalability and Load Balancing tests for PIPT 2.1

In the following tests, *Basetime* refers to the sequential execution time of the PIPT 2.1 routine and *Newtime* refers to the parallel execution time. All tests were carried out on the indicated computational environments. The software was compiled with Solaris C 4.2; the 140e and 170e machines were running Solaris 2.5.1; the Ultra Enterprise 3000 (E3000) was running Solaris 2.6.

Shared Memory Performance

Tables A.17 through A.22 show the PIPT performance tests for a strictly shared memory environment. In this case, all nodes are unloaded (except for the test program).

Configuration	# Threads	Basetime	Newtime	Speedup
1 Sun Ultra E3000	2	1.421	1.829	0.777
1 Sun Ultra E3000	4	1.421	1.779	0.799

Table A.17: PIPT performance of IPCenterMean on an SMP with an insignificant load average. Here, *Basetime* is the time obtained with a single thread and *Newtime* is the time obtained on the SMP with the indicated number of threads.

Configuration	# Threads	Basetime	Newtime	Speedup
1 Sun Ultra E3000	2	27.586	15.773	1.749
1 Sun Ultra E3000	4	27.586	8.099	3.406

Table A.18: PIPT performance of IPUnsharpMask on an SMP with an insignificant load average. Here, *Basetime* is the time obtained with a single thread and *Newtime* is the time obtained on the SMP with the indiciated number of threads.

Configuration	# Threads	Basetime	Newtime	Speedup
1 Sun Ultra E3000	2	4611.846	2553.691	1.806
1 Sun Ultra E3000	4	4611.846	1245.326	3.703

Table A.19: PIPT performance of IPCannyEdge on an SMP with an insignificant load average. Here, *Basetime* is the time obtained with a single thread and *Newtime* is the time obtained on the SMP with the indicated number of threads.

Configuration	**# Threads**	**Basetime**	**Newtime**	**Speedup**
1 Sun Ultra E3000	2	8341.226	4441.009	1.878
1 Sun Ultra E3000	4	8341.226	2087.790	3.995

Table A.20: PIPT performance of IPWindowMoments on an SMP with an insignificant load average. Here, *Basetime* is the time obtained with a single thread and *Newtime* is the time obtained on the SMP with the indicated number of threads.

Configuration	**# Threads**	**Basetime**	**Newtime**	**Speedup**
1 Sun Ultra E3000	2	29.225	16.447	1.777
1 Sun Ultra E3000	4	29.225	8.374	3.490

Table A.21: PIPT performance of IPAverage on an SMP with an insignificant load average. Here, *Basetime* is the time obtained with a single thread and *Newtime* is the time obtained on the SMP with the indiciated number of threads.

Configuration	**# Threads**	**Basetime**	**Newtime**	**Speedup**
1 Sun Ultra E3000	2	363.730	213.960	1.700
1 Sun Ultra E3000	4	363.730	107.480	3.384

Table A.22: PIPT performance of IPSquareMedian on an SMP with an insignificant load average. Here, *Basetime* is the time obtained with a single thread and *Newtime* is the time obtained on the SMP with the indicated number of threads.

Strictly Distributed Memory Performance in Dedicated Homogeneous Environment

Tables A.23 through A.28 show the PIPT parallel performance tests for a strictly distributed memory environment. In this case, all nodes are unloaded (except for the test program).

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	1.622	1.631	0.994
4 Sun UltraSPARC 140e	10bT	1.622	1.633	0.993
8 Sun UltraSPARC 140e	10bT	1.622	1.800	0.901
2 Sun UltraSPARC 140e	100bT	1.622	1.624	0.999
4 Sun UltraSPARC 140e	100bT	1.622	1.625	0.998
8 Sun UltraSPARC 140e	100bT	1.622	1.792	0.905

Table A.23: PIPT performance of IPCenterMean with all nodes having an insignificant load average. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	34.649	27.699	1.251
4 Sun UltraSPARC 140e	10bT	34.649	19.304	1.795
8 Sun UltraSPARC 140e	10bT	34.649	11.695	2.963
2 Sun UltraSPARC 140e	100bT	34.649	21.527	1.610
4 Sun UltraSPARC 140e	100bT	34.649	12.244	2.830
8 Sun UltraSPARC 140e	100bT	34.649	6.955	4.982

Table A.24: PIPT performance of IPUnsharpMask with all nodes having an insignificant load average. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	5479.237	3464.327	1.582
4 Sun UltraSPARC 140e	10bT	5479.237	2289.812	2.393
8 Sun UltraSPARC 140e	10bT	5479.237	1616.345	3.390
2 Sun UltraSPARC 140e	100bT	5479.237	3008.579	1.821
4 Sun UltraSPARC 140e	100bT	5479.237	1971.967	2.779
8 Sun UltraSPARC 140e	100bT	5479.237	1399.042	3.916

Table A.25: PIPT performance of IPCannyEdge with all nodes having an insignificant load average. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	10477.192	5910.164	1.773
4 Sun UltraSPARC 140e	10bT	10477.192	3185.516	3.289
8 Sun UltraSPARC 140e	10bT	10477.192	1802.212	5.814
2 Sun UltraSPARC 140e	100bT	10477.192	5480.389	1.912
4 Sun UltraSPARC 140e	100bT	10477.192	2964.529	3.534
8 Sun UltraSPARC 140e	100bT	10477.192	1708.583	6.132

Table A.26: PIPT performance of IPWindowMoments with all nodes having an insignificant load average. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	38.188	31.116	1.227
4 Sun UltraSPARC 140e	10bT	38.188	19.808	1.928
8 Sun UltraSPARC 140e	10bT	38.188	13.321	2.867
2 Sun UltraSPARC 140e	100bT	38.188	23.441	1.629
4 Sun UltraSPARC 140e	100bT	38.188	13.706	2.786
8 Sun UltraSPARC 140e	100bT	38.188	8.182	4.667

Table A.27: PIPT performance of IPAverage with all nodes having an insignificant load average. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	**Device**	**Basetime**	**Newtime**	**Speedup**
2 Sun UltraSPARC 140e	10bT	413.755	248.995	1.662
4 Sun UltraSPARC 140e	10bT	413.755	125.304	3.302
8 Sun UltraSPARC 140e	10bT	413.755	64.892	6.376
2 Sun UltraSPARC 140e	100bT	413.755	232.846	1.777
4 Sun UltraSPARC 140e	100bT	413.755	118.080	3.504
8 Sun UltraSPARC 140e	100bT	413.755	57.268	7.225

Table A.28: PIPT performance of IPSquareMedian with all nodes having an insignificant load average. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Strictly Distributed Memory Performance in Non-Dedicated Heterogeneous Environment

Tables A.29 through A.34 show the PIPT parallel performance tests for a strictly distributed memory environment. In this case, besides the test program, the indicated number of nodes are loaded with jobs that increase the load average to be approximately 1.0 in the absence of the test program.

Configuration	# Busy	Device	None	FFFS	RFFFS
2 Sun UltraSPARC 140e	1	10bT	1.629	1.628	1.630
4 Sun UltraSPARC 140e	2	10bT	1.713	1.628	1.629
8 Sun UltraSPARC 140e	4	10bT	1.809	1.626	1.632
2 Sun UltraSPARC 140e	1	100bT	1.626	1.629	1.624
4 Sun UltraSPARC 140e	2	100bT	1.634	1.626	1.626
8 Sun UltraSPARC 140e	4	100bT	1.642	1.627	1.629
1 Sun UltraSPARC 140e, 1 UltraSPARC 170e	1 (140e)	100bT	1.625	1.626	1.627
2 Sun UltraSPARC 140e, 2 UltraSPARC 170e	2 (140e)	100bT	1.627	1.626	1.627
4 Sun UltraSPARC 140e, 4 UltraSPARC 170e	4 (140e)	100bT	1.626	1.625	1.627

Table A.29: PIPT performance of IPCenterMean with some nodes having a significant load average (approximately 1.0), using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	# Busy	Device	None	FFFS	RFFFS
2 Sun UltraSPARC 140e	1	10bT	40.644	28.470	28.517
4 Sun UltraSPARC 140e	2	10bT	24.057	17.865	17.991
8 Sun UltraSPARC 140e	4	10bT	14.561	10.325	10.340
2 Sun UltraSPARC 140e	1	100bT	34.845	25.085	24.920
4 Sun UltraSPARC 140e	2	100bT	15.649	12.437	12.437
8 Sun UltraSPARC 140e	4	100bT	8.553	7.544	7.608
1 Sun UltraSPARC 140e, 1 UltraSPARC 170e	1 (140e)	100bT	33.119	23.486	23.480
2 Sun UltraSPARC 140e, 2 UltraSPARC 170e	2 (140e)	100bT	14.088	12.305	12.318
4 Sun UltraSPARC 140e, 4 UltraSPARC 170e	4 (140e)	100bT	8.481	6.972	7.037

Table A.30: PIPT performance of IPUnsharpMask with some nodes having a significant load average (approximately 1.0), using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	**# Busy**	**Device**	**None**	**FFFS**	**RFFFS**
2 Sun UltraSPARC 140e	1	10bT	5962.451	4851.562	4715.321
4 Sun UltraSPARC 140e	2	10bT	3674.529	3058.965	3105.447
8 Sun UltraSPARC 140e	4	10bT	2598.584	2186.254	2104.235
2 Sun UltraSPARC 140e	1	100bT	5265.683	4119.374	4198.276
4 Sun UltraSPARC 140e	2	100bT	3119.560	2528.598	2610.448
8 Sun UltraSPARC 140e	4	100bT	1935.123	1751.264	1780.788
1 Sun UltraSPARC 140e, 1 UltraSPARC 170e	1 (140e)	100bT	5174.964	3791.767	3744.408
2 Sun UltraSPARC 140e, 2 UltraSPARC 170e	2 (140e)	100bT	2971.776	2340.215	2413.681
4 Sun UltraSPARC 140e, 4 UltraSPARC 170e	4 (140e)	100bT	1888.533	1641.583	1696.345

Table A.31: PIPT performance of IPCannyEdge with some nodes having a significant load average (approximately 1.0), using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	**# Busy**	**Device**	**None**	**FFFS**	**RFFFS**
2 Sun UltraSPARC 140e	1	10bT	10012.239	6998.518	6845.258
4 Sun UltraSPARC 140e	2	10bT	4987.830	3675.310	3712.534
8 Sun UltraSPARC 140e	4	10bT	2615.547	2085.654	2098.563
2 Sun UltraSPARC 140e	1	100bT	8999.621	6819.180	6743.561
4 Sun UltraSPARC 140e	2	100bT	4522.571	3576.960	3586.012
8 Sun UltraSPARC 140e	4	100bT	2417.306	1940.014	2016.192
1 Sun UltraSPARC 140e, 1 UltraSPARC 170e	1 (140e)	100bT	8907.163	6334.656	6185.544
2 Sun UltraSPARC 140e, 2 UltraSPARC 170e	2 (140e)	100bT	4482.413	3275.263	3186.548
4 Sun UltraSPARC 140e, 4 UltraSPARC 170e	4 (140e)	100bT	2408.607	1895.320	1906.728

Table A.32: PIPT performance of IPWindowMoments with some nodes having a significant load average (approximately 1.0), using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	# Busy	Device	None	FFFS	RFFFS
2 Sun UltraSPARC 140e	1	10bT	52.303	37.295	37.317
4 Sun UltraSPARC 140e	2	10bT	31.858	24.240	24.300
8 Sun UltraSPARC 140e	4	10bT	16.524	12.159	11.893
2 Sun UltraSPARC 140e	1	100bT	42.602	26.833	26.689
4 Sun UltraSPARC 140e	2	100bT	19.008	13.903	13.951
8 Sun UltraSPARC 140e	4	100bT	9.638	8.754	7.749
1 Sun UltraSPARC 140e, 1 UltraSPARC 170e	1 (140e)	100bT	34.902	24.291	24.392
2 Sun UltraSPARC 140e, 2 UltraSPARC 170e	2 (140e)	100bT	18.762	12.776	12.654
4 Sun UltraSPARC 140e, 4 UltraSPARC 170e	4 (140e)	100bT	9.007	7.533	7.550

Table A.33: PIPT performance of IPAverage with some nodes having a significant load average (approximately 1.0), using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	# Busy	Device	None	FFFS	RFFFS
2 Sun UltraSPARC 140e	1	10bT	457.627	304.362	303.569
4 Sun UltraSPARC 140e	2	10bT	235.000	154.486	155.321
8 Sun UltraSPARC 140e	4	10bT	125.413	88.974	88.249
2 Sun UltraSPARC 140e	1	100bT	411.434	299.269	295.466
4 Sun UltraSPARC 140e	2	100bT	198.898	148.119	148.355
8 Sun UltraSPARC 140e	4	100bT	101.164	74.884	74.830
1 Sun UltraSPARC 140e, 1 UltraSPARC 170e	1 (140e)	100bT	402.022	274.567	267.158
2 Sun UltraSPARC 140e, 2 UltraSPARC 170e	2 (140e)	100bT	191.615	132.620	133.526
4 Sun UltraSPARC 140e, 4 UltraSPARC 170e	4 (140e)	100bT	100.941	66.844	66.803

Table A.34: PIPT performance of IPSquareMedian with some nodes having a significant load average (approximately 1.0), using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Mixed Shared and Distributed Memory Performance in Non-Dedicated Heterogeneous Environment

Tables A.35 through A.40 show the PIPT parallel performance tests for a strictly distributed memory environment. In this case, besides the test program the indicated nodes are loaded with jobs that increase the load average to be approximately 1.0 in the absence of the test program.

Configuration	# Busy	Device	None	FFFS	RFFFS
1 Sun Ultra E3000, 1 Sun UltraSPARC 140e	1 (140e)	100bT	1.676	1.658	1.652
1 Sun Ultra E3000, 3 Sun UltraSPARC 140e	2 (140e)	100bT	1.678	1.647	1.521
1 Sun Ultra E3000, 7 Sun UltraSPARC 140e	4 (140e)	100bT	1.679	1.660	1.496
1 Sun Ultra E3000, 1 Sun UltraSPARC 170e	1 (170e)	100bT	1.629	1.628	1.628
1 Sun Ultra E3000, 3 Sun UltraSPARC 170e	2 (170e)	100bT	1.641	1.635	1.638

Table A.35: PIPT performance on IPCenterMean on workstation cluster with an SMP, with some nodes having a significant load average, using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	# Busy	Device	None	FFFS	RFFFS
1 Sun Ultra E3000, 1 Sun UltraSPARC 140e	1 (140e)	100bT	38.510	25.318	25.484
1 Sun Ultra E3000, 3 Sun UltraSPARC 140e	2 (140e)	100bT	20.737	14.480	13.890
1 Sun Ultra E3000, 7 Sun UltraSPARC 140e	4 (140e)	100bT	11.303	9.234	9.183
1 Sun Ultra E3000, 1 Sun UltraSPARC 170e	1 (170e)	100bT	35.571	24.812	24.706
1 Sun Ultra E3000, 3 Sun UltraSPARC 170e	2 (170e)	100bT	18.865	13.116	12.595

Table A.36: PIPT performance of IPUnsharpMask on workstation cluster with an SMP, with some nodes having a significant load average, using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	# Busy	Device	None	FFFS	RFFFS
1 Sun Ultra E3000, 1 Sun UltraSPARC 140e	1 (140e)	100bT	4924.127	2707.897	2415.663
1 Sun Ultra E3000, 3 Sun UltraSPARC 140e	2 (140e)	100bT	2731.889	2076.864	1834.934
1 Sun Ultra E3000, 7 Sun UltraSPARC 140e	4 (140e)	100bT	1648.163	1364.681	1385.660
1 Sun Ultra E3000, 1 Sun UltraSPARC 170e	1 (170e)	100bT	4466.876	2111.496	1906.118
1 Sun Ultra E3000, 3 Sun UltraSPARC 170e	2 (170e)	100bT	2326.119	1614.779	1486.334

Table A.37: PIPT performance of IPCannyEdge on workstation cluster with an SMP, with some nodes having a significant load average, using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	# Busy	Device	None	FFFS	RFFFS
1 Sun Ultra E3000, 1 Sun UltraSPARC 140e	1 (140e)	100bT	8707.118	4086.896	3887.989
1 Sun Ultra E3000, 3 Sun UltraSPARC 140e	2 (140e)	100bT	4385.833	2899.406	2285.361
1 Sun Ultra E3000, 7 Sun UltraSPARC 140e	4 (140e)	100bT	2288.131	1485.716	1440.231
1 Sun Ultra E3000, 1 Sun UltraSPARC 170e	1 (170e)	100bT	7683.495	3606.434	3430.911
1 Sun Ultra E3000, 3 Sun UltraSPARC 170e	2 (170e)	100bT	3870.227	2558.547	2016.690

Table A.38: PIPT performance of IPWindowMoments on workstation cluster with an SMP, with some nodes having a significant load average, using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	# Busy	Device	None	FFFS	RFFFS
1 Sun Ultra E3000, 1 Sun UltraSPARC 140e	1 (140e)	100bT	41.734	26.086	25.819
1 Sun Ultra E3000, 3 Sun UltraSPARC 140e	2 (140e)	100bT	21.938	15.471	15.610
1 Sun Ultra E3000, 7 Sun UltraSPARC 140e	4 (140e)	100bT	12.230	11.014	8.671
1 Sun Ultra E3000, 1 Sun UltraSPARC 170e	1 (170e)	100bT	38.802	25.128	24.997
1 Sun Ultra E3000, 3 Sun UltraSPARC 170e	2 (170e)	100bT	20.116	14.825	14.765

Table A.39: PIPT performance of IPAverage on workstation cluster with an SMP, with some nodes having a significant load average, using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

Configuration	**# Busy**	**Device**	**None**	**FFFS**	**RFFFS**
1 Sun Ultra E3000, 1 Sun UltraSPARC 140e	1 (140e)	100bT	399.518	197.213	150.848
1 Sun Ultra E3000, 3 Sun UltraSPARC 140e	2 (140e)	100bT	200.602	131.244	101.324
1 Sun Ultra E3000, 7 Sun UltraSPARC 140e	4 (140e)	100bT	103.753	67.649	65.595
1 Sun Ultra E3000, 1 Sun UltraSPARC 170e	1 (170e)	100bT	355.920	148.194	130.054
1 Sun Ultra E3000, 3 Sun UltraSPARC 170e	2 (170e)	100bT	179.664	87.115	75.219

Table A.40: PIPT performance of IPSquareMedian on workstation cluster with an SMP, with some nodes having a significant load average, using no load balancing (*None*), First-Finished, First-Served (*FFFS*), and Redundant First-Finished, First-Served (*RFFFS*).

A.4.3 Parallel HRVS

The performance testing program takes the Notre Dame Administration Building video sequence as input, and uses the PHRVS to generate the expanded output image. The "Dome" video sequence consists of three color (three-plane) 640×480 images, and is shown as Figure A.5 in Appendix A.5. The PHRVS is tested in two environments: 1, 2, 4, and 8 worker UltraSPARC 140E nodes, and 1, 2, and 4 worker threads on a single UltraSPARC Enterprise 3000 server.

Table A.41 shows the results of these timings. *Basetime* is the serial time in each environment, while *Newtime* is the time for multiple nodes/threads.

Configuration	**Device**	**Basetime**	**Newtime**	**Speedup**
1 Sun UltraSPARC 140e	100bT	2280.800	2280.800	1.000
2 Sun UltraSPARC 140e	100bT	2280.800	1160.300	1.966
4 Sun UltraSPARC 140e	100bT	2280.800	662.100	3.445
8 Sun UltraSPARC 140e	100bT	2280.800	356.100	6.405
1 Sun Ultra E3000	1 thread	1963.700	1963.700	1.000
1 Sun Ultra E3000	2 threads	1963.700	994.700	1.974
1 Sun Ultra E3000	4 threads	1963.700	575.500	3.412

Table A.41: PHRVS performance with all nodes having an insignificant load average

Other testing is also performed to show the preformance gain by using VIS (Visual Instruction Set). Both the test images and configuration are the same as the previous test.

Table A.42 shows the results of these timings. *Basetime* is PHRVS run time without VIS, while *Newtime* is the PHRVS run time with VIS.

Configuration	**Device**	**Basetime**	**Newtime**	**Speedup**
1 Sun UltraSPARC 140e	100bT	6506.400	2280.800	2.853
2 Sun UltraSPARC 140e	100bT	3309.900	1160.300	2.853
4 Sun UltraSPARC 140e	100bT	1642.800	662.100	2.481
8 Sun UltraSPARC 140e	100bT	754.600	356.100	2.119
1 Sun Ultra E3000	1 thread	5845.000	1963.700	2.977
1 Sun Ultra E3000	2 threads	2808.500	994.700	2.824
1 Sun Ultra E3000	4 threads	1374.000	575.500	2.388

Table A.42: PHRVS performance with all nodes having an insignificant load average

A.4.4 Parallel Visualization

Performance testing was also done on the PVIZ toolkit to demonstrate speed up due to multi-threading on multiple processors. A test program and timing results for one and four processors was produced. Each of the PVIZ functions was timed to demonstrate the performance increase due to multi-threaded processing.

Table A.43 shows the results of these timings. *Basetime* is the serial (i.e., a single thread) time, while *Newtime* is the time for four threads. The image processed was `toys.tif`, a color 512 × 512 image, and is shown as Figure A.4 in Appendix A.5.

Function	**Basetime**	**Newtime**	**Speedup**
`PVIZ_ConvolveII`	0.747	0.388	1.925
`PVIZ_ConvolveID`	0.683	0.272	2.511
`PVIZ_ConvolveDI`	0.712	0.340	2.094
`PVIZ_ConvolveDD`	0.646	0.241	2.680
`PVIZ_CopyImageII`	0.417	0.327	1.275
`PVIZ_CopyImageID`	0.466	0.252	1.849
`PVIZ_CopyImageDI`	0.430	0.293	1.468
`PVIZ_CopyImageDD`	0.364	0.188	1.936
`PVIZ_EdgeDetectionII`	1.138	0.403	2.824
`PVIZ_EdgeDetectionID`	1.099	0.287	3.829
`PVIZ_EdgeDetectionDI`	1.145	0.380	3.013
`PVIZ_EdgeDetectionDD`	1.072	0.254	4.220
`PVIZ_RotateII`	0.477	0.336	1.420
`PVIZ_RotateID`	0.544	0.261	2.084
`PVIZ_RotateDI`	0.425	0.307	1.384
`PVIZ_RotateDD`	0.482	0.239	2.017
`PVIZ_ScaleII`	0.479	0.325	1.478
`PVIZ_ScaleID`	0.417	0.227	1.837
`PVIZ_ScaleDI`	0.430	0.294	1.463
`PVIZ_ScaleDD`	0.365	0.184	1.984
`PVIZ_ErodeII`	1.352	0.608	2.224
`PVIZ_ErodeID`	1.295	0.504	2.569
`PVIZ_ErodeDI`	1.300	0.569	2.285
`PVIZ_ErodeDD`	1.248	0.469	2.661

Table A.43: PVIZ performance on an unloaded Sun Ultra-Entrprise 3000 Server. Here, *Basetime* is the time obtained on a Sun Ultra 170E using XIL 1.2 and *Newtime* is the time obtained on a four processor Sun Ultra E3000 using XIL 1.3.

A.4.5 Combining Manager and Worker on a Single Node

Tables A.44 through A.49 show the PIPT parallel performance tests for a strictly distributed memory environment. In this case, all nodes are unloaded (except for the test program) and the manager process is run on a node with one of the worker processes.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	1.622	1.715	0.946
4 Sun UltraSPARC 140e	10bT	1.622	1.716	0.945
8 Sun UltraSPARC 140e	10bT	1.622	1.724	0.941
2 Sun UltraSPARC 140e	100bT	1.622	1.621	1.001
4 Sun UltraSPARC 140e	100bT	1.622	1.621	1.001
8 Sun UltraSPARC 140e	100bT	1.622	1.629	0.996

Table A.44: PIPT performance of IPCenterMean with all nodes having an insignificant load average, with the manager process running on the same node as one of the worker processes. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	34.649	28.119	1.232
4 Sun UltraSPARC 140e	10bT	34.649	22.287	1.555
8 Sun UltraSPARC 140e	10bT	34.649	16.241	2.133
2 Sun UltraSPARC 140e	100bT	34.649	22.481	1.541
4 Sun UltraSPARC 140e	100bT	34.649	11.396	3.040
8 Sun UltraSPARC 140e	100bT	34.649	7.089	4.888

Table A.45: PIPT performance of IPUnsharpMask with all nodes having an insignificant load average, with the manager process running on the same node as one of the worker processes. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	5479.237	4631.612	1.183
4 Sun UltraSPARC 140e	10bT	5479.237	2675.502	2.048
8 Sun UltraSPARC 140e	10bT	5479.237	1797.584	4.282
2 Sun UltraSPARC 140e	100bT	5479.237	3630.532	1.509
4 Sun UltraSPARC 140e	100bT	5479.237	2173.525	2.521
8 Sun UltraSPARC 140e	100bT	5479.237	1491.259	3.674

Table A.46: PIPT performance of IPCannyEdge with all nodes having an insignificant load average, with the manager process running on the same node as one of the worker processes. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	10477.192	7841.304	1.336
4 Sun UltraSPARC 140e	10bT	10477.192	5003.123	2.094
8 Sun UltraSPARC 140e	10bT	10477.192	2446.959	4.282
2 Sun UltraSPARC 140e	100bT	10477.192	5617.601	1.865
4 Sun UltraSPARC 140e	100bT	10477.192	2891.444	3.624
8 Sun UltraSPARC 140e	100bT	10477.192	1587.261	6.601

Table A.47: PIPT performance of IPWindowMoments with all nodes having an insignificant load average, with the manager process running on the same node as one of the worker processes. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	Device	Basetime	Newtime	Speedup
2 Sun UltraSPARC 140e	10bT	38.188	28.284	1.350
4 Sun UltraSPARC 140e	10bT	38.188	22.372	1.707
8 Sun UltraSPARC 140e	10bT	38.188	17.032	2.242
2 Sun UltraSPARC 140e	100bT	38.188	25.318	1.508
4 Sun UltraSPARC 140e	100bT	38.188	14.115	2.705
8 Sun UltraSPARC 140e	100bT	38.188	7.104	5.376

Table A.48: PIPT performance of IPAverage with all nodes having an insignificant load average, with the manager process running on the same node as one of the worker processes. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

Configuration	**Device**	**Basetime**	**Newtime**	**Speedup**
2 Sun UltraSPARC 140e	10bT	413.755	321.863	1.286
4 Sun UltraSPARC 140e	10bT	413.755	214.250	1.931
8 Sun UltraSPARC 140e	10bT	413.755	106.144	3.898
2 Sun UltraSPARC 140e	100bT	413.755	233.846	1.769
4 Sun UltraSPARC 140e	100bT	413.755	118.803	3.483
8 Sun UltraSPARC 140e	100bT	413.755	58.784	7.039

Table A.49: PIPT performance of IPSquareMedian with all nodes having an insignificant load average, with the manager process running on the same node as one of the worker processes. Here, *Basetime* is the time obtained on a single Sun Ultra 140E and *Newtime* is the time obtained by the indicated cluster.

A.5 Test Suite Input Images

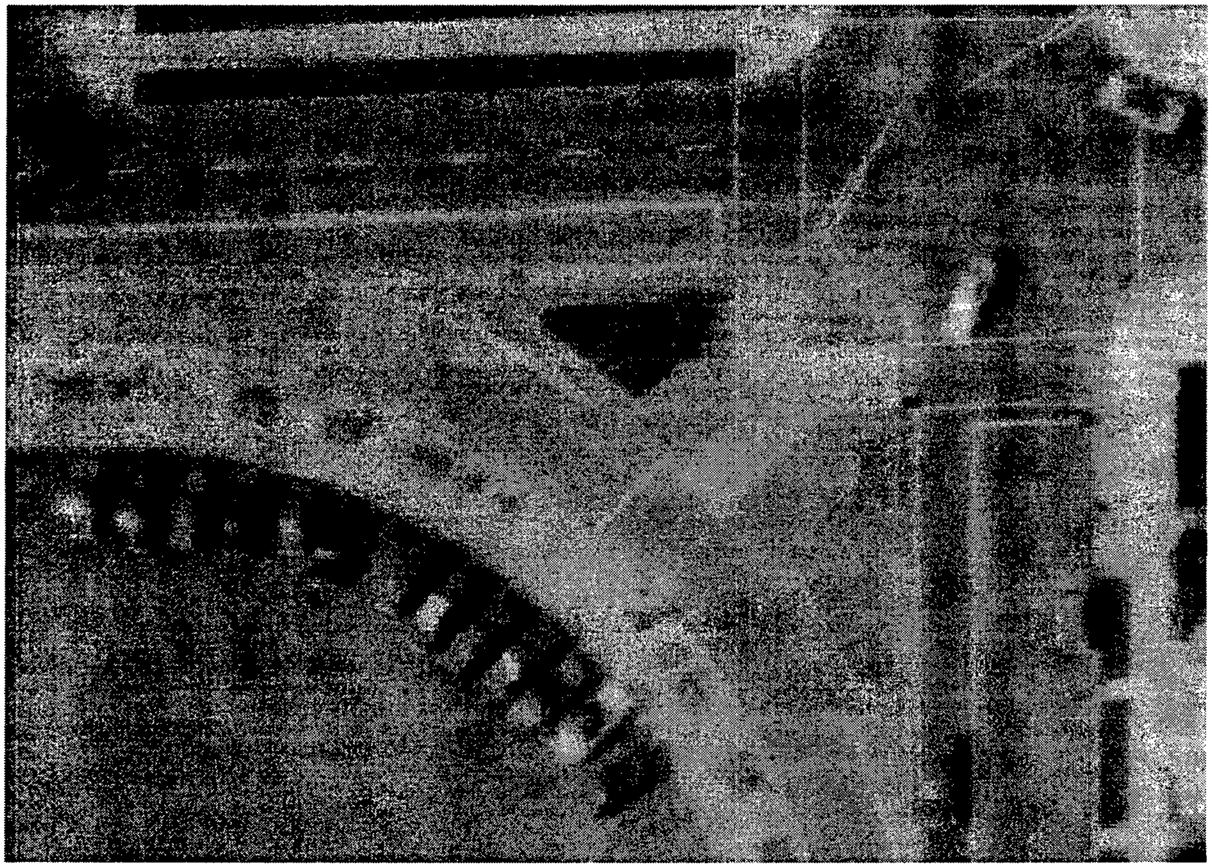


Figure A.1: `cars.tif`, single plane black and white 896 × 636 test image (shown half size) used for PIPT functionality testing

Figure A.2: `eggs.tif`, three plane color 258 × 220 test image (shown half size) used for PIPT functionality testing

Figure A.3: `big.tif`, one plane black and white 2910 × 3080 test image (shown one eighth of original size) used for PIPT performance testing

Figure A.4: `cars.tif`, three plane color 512 × 512 test image (shown half size) used for PVIZ functionality and performance testing

Figure A.5: Dome sequence, three three-plane color 640 × 480 test images (each shown half size) used for PHRVS functionality and performance testing

MISSION OF AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.